FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Real-Time Ray Traced Voxel Global Illumination

Tiago Magalhães

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Jacob

Supervisor: Teresa Matos

July 30, 2020

Real-Time Ray Traced Voxel Global Illumination

Tiago Magalhães

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Gilberto Bernardes
External Examiner: António Fernandes
Supervisor: João Jacob

July 30, 2020

Abstract

The real-time rendering industry currently finds itself at a crossroads. Recent advances in graphics processing unit hardware powered by Nvidia have allowed us to reach a long sought after goal of being able to render ray-traced images in real-time; with this, with framerates of 60 or higher being achievable in modern and recent video games on top-of-the-line hardware. Despite this great advancement, there are still two core issues; this rendering process is still much more expensive than conventional rasterization, the same scene could have been many times faster to render if ray tracing was not used; the other issue is that with this hardware being very new, there is a high early adopter price to be paid by consumers, in consequence slowing down the adoption of this technology by companies.

To combat these issues there is an ongoing effort to bridge the gap between ray tracing and rasterization to bring ray tracing to the masses with as little impact on rendering time as possible, these techniques are called hybrid rendering techniques.

This work aims to combat one big issue introduced with real-time ray tracing, resolution. Currently, on top-of-the-line hardware 1920x1080 tends to be the maximum achievable video output resolution when using ray tracing. As such this work presents a technique for global illumination that aims to decouple the video output resolution from ray tracing resolution. This technique combines existing research into voxel-based global illumination with the new hardware-accelerated ray tracing technology to achieve its goal. The presented technique is capable of handling both diffuse and specular illumination, as well as producing soft-shadows. Under the correct configuration, the presented technique is capable of globally illuminating a scene in under 2ms at 1920x1080 in scenes like Sponza and Sibenik.

Acknowledgments

I would like to take a moment to acknowledge those who have greatly contributed to the existence of this work. First and foremost, I would like to thank my supervisors for guiding me through this process as well as thoroughly reviewing all the content in this work. I would like to thank my family which provided the support necessary for this work's existence and for the academic journey that led to this work. I would like to thank all those involved in the creation of the book Real-Time Rendering[1] which was an invaluable resource to this work's development. Lastly, I would like to thank the Faculty of Engineering of the University of Porto for allowing me to produce this work and for providing me with the necessary knowledge and skillset to tackle such an endeavor.

Contents

List of Figures

List of Tables

Abbreviations

RTX	Nvidia's hardware-accelerated GPU ray tracing technology
D3D12	Direct3D 12
BRDF	Bidirectional Reflectance Distribution Function
sRGB	Standard red green blue
GPU	Graphics processing unit
DXR	DirectX ray tracing
API	Application programming interface
BVH	Bounding-Volume hierarchy
CPU	Centra processing unit
LOD	Level-of-Detail
MSAA	Multisample antialiasing
BC1-7	Block compression texture format 1 through 7
VXGI	Voxel-accelerated global illumination
SDK	Software development kit
COM	Component object model
UAV	Unordered access view
SRV	Shader resource view
HLSL	High-level shading language
TLAS	Top-level acceleration structure
BLAS	Bottom-level acceleration structure
SSIM	Structural similarity index measure
MSVC	Microsoft Visual C++
VRAM	Video Random Access Memory
HVS	Human visual system
VXAO	Voxel-accelerated ambient occlusion
TAA	Temporal antialiasing

Chapter 1

Introduction

In the field of real-time rendering, ray tracing is seen as the holy grail, being capable of producing stunning images compared to the traditional approach of rasterization. However this comes at a great performance cost, and as such the field has for the longest time had the goal of achieving real-time ray tracing.

Recent advances in GPU hardware have allowed us to achieve the goal of real-time ray tracing. That being said, due to the recency of this technology, the amount of public research into this subject rather scarce. Additionally, implementations of this technology at this time have a significant weight on frame render times.

In this chapter, you will find a historical contextualization (Section 1.1) and motivation (Section 1.2) of this work, its objectives (Section 1.3), and the structure of this document (Section 1.4) as a whole.

1.1 Context

Computer graphics the two dominant rendering approaches are rasterization and ray tracing. Rasterization works by mapping objects in a 3D scene to the screen, this, however, is an unnatural way to model lighting and leads to the need for specific models for each visual effect that we want to render. On the other hand, Ray tracing and its stochastic variation Path tracing, model how light moves, in reality, leading to a much more natural way to model lighting, capable of describing a wide variety of effects on its own. As a consequence ray tracing produces much better looking and more realistic images when compared to rasterization (Figure 3.3).

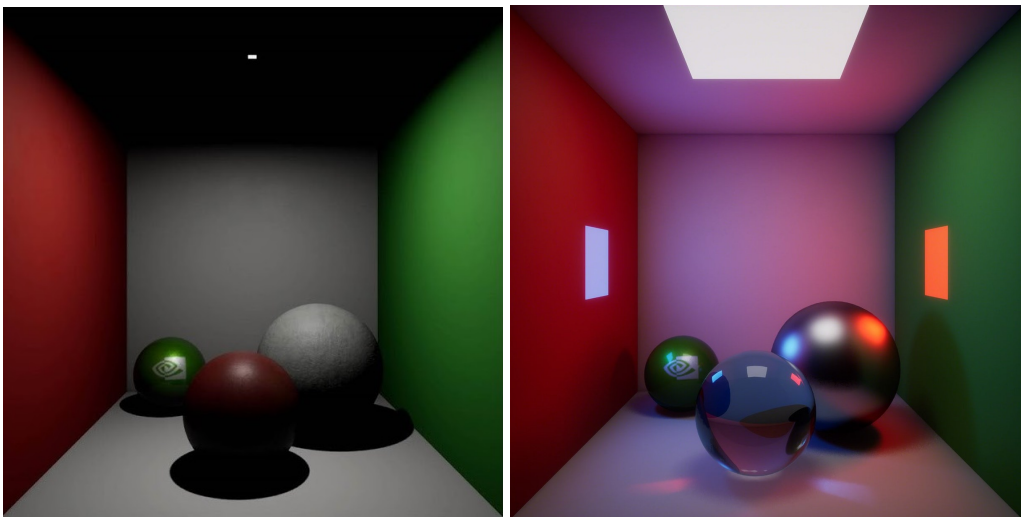

Figure 1.1: An image comparing rasterization (left) and ray tracing (right) used in the cornel box. These images have been provided by Nvidia.

For decades, the field of real-time rendering has strived to achieve real-time ray tracing. Some believe that just like rasterization, attaining real-time ray tracing in complex scenes requires the aid of specialized hardware. Real-time renderers have relied on clever approximations and precomputing data to emulate the results that a ray tracing renderers can achieve. While efficient and sometimes visually convincing, these methods have always presented corner case issues, while also never fully emulating the quality of ray tracing.

In August 2018, Nvidia officially announced their Turing architecture GPUs, these came with one fundamental breakthrough: hardware-accelerated ray tracing. For the first time, complex high-quality scenes could be rendered in real-time at resolutions of 1920x1080 or more. This new technology has paved the way for a fundamental paradigm shift in the field of real-time rendering.

Kajiya[23] introduced the rendering equation, this equation has become the standard way that we describe how light behaves in a scene. The rendering equation tells us that the light that leaves a surface is a sum of the light it emits and a part of the light that reaches it. However, this equation is not usable in practice due to its recursive nature. Ray tracing can model this recursive effect very elegantly despite it not being possible to allow it to be fully recursive, leading to highly realistic results as seen in Figure 3.3. However, ray tracing is highly demanding, introducing a significant increase in render times. Due to this nature, ray tracing has been isolated to offline(ahead-of-time) rendering, with real-time renderers focusing on approximating ray tracing or relying on pre-computed data from offline renderers to also approximate ray tracing's results.

Recent evolutions of graphics APIs have helped to address the high-performance requirements of ray tracing, focusing on high-performance rendering with multithreaded rendering and allowing developers to take full control over resources. Without these fundamental shifts in graphics software development, it would be significantly harder to achieve real-time ray tracing, and its impact on render time could be much more significant.

It is now 2020 and despite the recency of this technology, real-time ray tracing has been featured in several renowned game franchises in both 2018 and 2019. Games such as Call of Duty Modern Warfare and Control (Fig 1.2) featured real-time ray tracing support on release. This relatively quick adoption by some renowned game studios supports the notion that real-time ray tracing is not a gimmick, but rather a revolutionary technology that may shape the future of real-time rendering.

The 9th generation of home consoles, which is releasing at the end of 2020, has also publicly committed to supporting hardware-accelerated ray tracing, further emphasizing the importance of the paradigm shift this technology brings to real-time rendering.

Figure 1.2: Images from Control and Metro Exodus, video games that utilize real-time ray tracing technology.

An innovative way in which real-time ray tracing was used since its release is to renovate classic games. Nvidia remastered the classic game Quake II, introducing built-in real-time ray tracing support, releasing this updated version under the name of Quake II RTX (Fig 1.3). A continuation of this trend would allow classic games to have a new life and reach audiences that might not have tried them due to their outdated visuals.

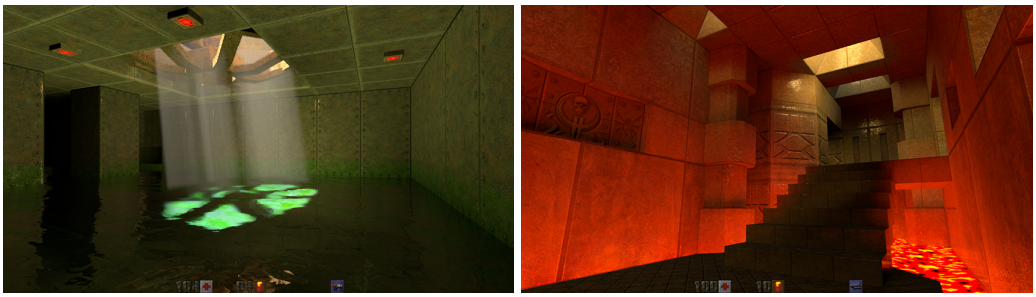

Figure 1.3: Screenshots of Quake II RTX, a modernized version of the classic game Quake II, released by Nvidia to showcase their RTX real-time ray tracing technology.

A field where the adoption of real-time ray tracing can be of significant value is that of digital artistic tooling, especially for VFX artists. Tools utilized in this field utilize real-time rendering to give the artist instant feedback, constraining offline rendering to only when the artist requires it. Integrating real-time ray tracing would allow artists to obtain better and more accurate instant feedback, reducing the number of times they are required to offline render their work to verify how it would look as a final product.

In summary, the adoption of real-time ray tracing can have benefits to several industries and different types of users, and as such fully adopting this technology and perfecting it has great value.

1.2 Motivation

While the achievement of real-time ray tracing represents a significant advancement in and of itself, there are still issues preventing it from achieving widespread mainstream adoption. Nvidia has released performance benchmarking results on some video games that have implemented real-time ray tracing effects (Table 1.1). These benchmarks were run at a resolution of 1920x1080 on a computer with a Core i7-5960X CPU, an RTX 2080 Ti GPU, 64GB of RAM and running on a 64-Bit Windows 10 operating system. These results show that, while usable, ray tracing still has a significant cost in frame render time. Two important aspects to consider with these results is that they were running on top-of-the-line Nvidia RTX 2080 Ti graphics card and that the resolution at which rays are cast might be significantly lower than the output image resolution. We can, therefore, argue that ray tracing still requires a significant amount of investment into research and development before it can achieve widespread mainstream adoption.

Over the decades, a high percentage of research into real-time rendering has focused on approximating the results of ray tracing, while comparatively, the amount of work that utilizes ray tracing directly is much more scarce. This scarcity mainly comes from the impracticality of using ray tracing in practice in a real-time system, however, the recency of this technology also contributes to this lack of research. In 2018, the advent of hardware-accelerated ray tracing overthrew the impracticality of ray tracing's usage in real-time systems.

In less than a year, the 9th generation of home consoles will be released. Both Microsoft and Sony have committed to supporting hardware-accelerated ray tracing with

Benchmark Name	RTX Off (ms)	RTX On (ms)	Delta (ms)
Call Of Duty: Modern Warfare Ray Traced Shadows	10.26694045	12.45330012	2.186359673
Control Ray Traced Reflections	13.73626374	16.55629139	2.820027654
Control Ray Traced Transparent Reflections	8.680555556	11.0619469	2.381391347
Control Ray Traced Indirect Diffuse Lighting	8.748906387	10.08064516	1.331738775
Control Ray Traced Debris (All Other RTX Effects Are On)	19.84126984	21.41327623	1.57200639
Control Ray Traced Contact Shadows	9.033423668	12.43781095	3.404387278

Table 1.1: Real-time ray tracing(RTX) performance benchmark presented by Nvidia. These benchmarks compare the frame rendering time with RTX enable and disabled.

their 9th generation consoles, which is likely to lead to an increased adoption rate of real-time ray tracing in the field of video games. Consoles represent a significant percentage of video game market share, leading developers to focus on these platforms, which results in consoles frequently dictating the pace of technological adoption. Consoles tend to be much more limited hardware-wise than their PC counterparts, and this restriction translates either into the usage of highly efficient algorithms or significant image quality tradeoffs. These restrictions lead to the conclusion that real-time ray tracing's adoption rate will be dependent on the efficiency of the algorithms that utilize it.

This work results from the belief that there is a need to invest in optimizing existing techniques to take full advantage of hardware-accelerated ray tracing and into developing new methods based on the existence of hardware-accelerated ray-tracing. That being said, currently, only Nvidia offers hardware-accelerated ray tracing with their Turing Architecture, as such this work is fully based on this hardware platform's characteristics.

1.3 Objectives

This work aims to develop a method for rendering globally illuminated images in real-time, at a relatively low performance cost. Additionally, this work will present a performance and image quality analysis of the developed technique. The following list outlines the objectives of this document.

- Conduct a state-of-the-art review of real-time global illumination rendering and image quality analysis.
- Develop a real-time hybrid global illumination method that blends voxelization and ray tracing.
- Test the developed method and conduct a performance and image quality analysis of this method's implementation.

1.4 Document Structure

Following this introductory chapter, this document presents five additional chapters.

Chapter 2 proceeds the current chapter, it presents a state-of-the-art review of computer graphics techniques focusing on the real-time simulation of global illumination, as well as a state-of-the-art review on user image perception in interactive environments.

Chapter 3 presents a proposal of the methodology that must be utilized to meet this work's goals and important considerations.

Chapter 4 presents an implementation of the proposed solution, focusing on implementation details that vary significantly from API to API and that may deviate from the solution architecture.

Chapter 5 presents an image quality and performance analysis of the results generated by the solution implementation.

Chapter 6 concludes this document by summarizing the author's conclusions regarding the proposed method, lessons learned, and a proposition of future work on the proposed methodology.

Chapter 2

State Of The Art

To develop a hybrid rendering method, we must first understand what the state of the art in the field it is inserted, and on top of that, we must understand a way to validate the results of our method. This chapter presents a state-of-the-art review of real-time global illumination rendering and a state-of-the-art review of image quality analysis.

2.1 Real-Time Global Illumination Rendering

Global illumination in real-time computer graphics is a very deep and complex subject, having gone through many evolutions over the decades. In this section, we go over the primary ways currently used to simulate global illumination. These include ambient occlusion, directional occlusion, diffuse global illumination, and specular global illumination.

2.1.1 Introductory Concepts

The field of computer graphics is vast and complex, most methods rely on prior knowledge about the field. This work like many others builds upon previous work, however, this means it also relies on prior knowledge about the field, as such this section will provide an introduction to many of the concepts that are assumed as prior knowledge in the rest of this work.

2.1.1.1 Bidirectional Reflectance Distribution Function - BRDF

The bidirectional reflectance distribution function or BRDF for short defines how a given surface reflects incoming light. This function varies with each specific surface and its properties as well as with approach to surface modeling. Mathematically this function

is usually represented as a function that received an incoming light direction and a view direction and returns a floating-point value representing the fraction that incident light that is reflected in the view direction.

2.1.1.2 Lambertian Surfaces

Lambertian surfaces are a special kind of diffuse surface, they reflect light uniformity in all directions, also known as a perfectly diffuse surface. Since it distributes light uniformly, its BRDF is a constant function, usually defined as:

$$f(l,v) = \frac{\rho_{ss}}{\pi} \tag{2.1}$$

where ρ_{ss} is called the subsurface albedo and is a floating-point value representing the percentage of the incoming light that the surface reflects.

Since this function is constant, it has the special characteristic of being able to be pulled from integrals that utilize a BRDF, this will become important in further sections as being able to pull the BRDF from an integral can represent a significant speedup in some cases.

2.1.1.3 Reflectance Equation

The reflectance equation standerdizes the light that is reflected off a given point, given a view direction, the surface BRDF and the light incoming into the surface. It is usually defined as:

$$L_o(p,v) = \int_{l \in \Omega} f(l,v) L_i(p,l)(n \cdot l)^+ dl \tag{2.2}$$

where:

- $L_o(p,v)$ is a floating-point value representing the light that is reflected from a position p given the view direction v
- Ω is the set of 3D direction from which light hits the surface, also called the surfaces hemisphere
- l is a 3D vector representing one of the direcion in the Ω set
- $f(l,v)$ is a function which returns a floating-point value and represents the surfaces BRDF
- $L_i(p,l)$ is a function which returns a floating point value and represents the incoming light into position p from direction l

- n is a 3D vector, representin the surface normal

- $(n \cdot l)^+$ is the cosine weighting factor

2.1.1.4 Cosine Weighting

Light that hits a surface in a direction closer to its normal will contribute to surface lighting much more than light rays that hit the surface from directions which are far from the surface normal. This effect is modeled through cosine weighting, mathmatically this is done through the dot product between the surface normal and the incoming light direction, the cosine term of the dot product is the therefore the origin of the name cosine weighting as that is the term that is responsible for this effect. Cosine wheighting is also used to model view depend effects such as specular reflections.

2.1.2 Ambient Occlusion

Ambient occlusion is one of the earliest attempts to approximate global illumination. It simulates how light that would hit a given surface is occluded by other objects in the scene. It was introduced by *Landis*[31] at Industrial Light and Magic in 2002, and in its initial formulation, this technique represented a significant oversimplification, however, it produced convincing results.

This section succinctly goes over both the theory behind ambient occlusion and the various categories of methods utilized to implement it in a real-time setting.

2.1.2.1 Theory of Ambient Occlusion

The principal idea behind this approach is that for a given surface point we have a hemisphere with all the directions from which light can come. For each of the incoming light directions, occlusion is checked and the individual contributions are added up (through integration) to calculate the actual light that hits a point (Figure 2.1). These occlusion checks are performed with a visibility function, which can vary depending on the implementation and the desired results.

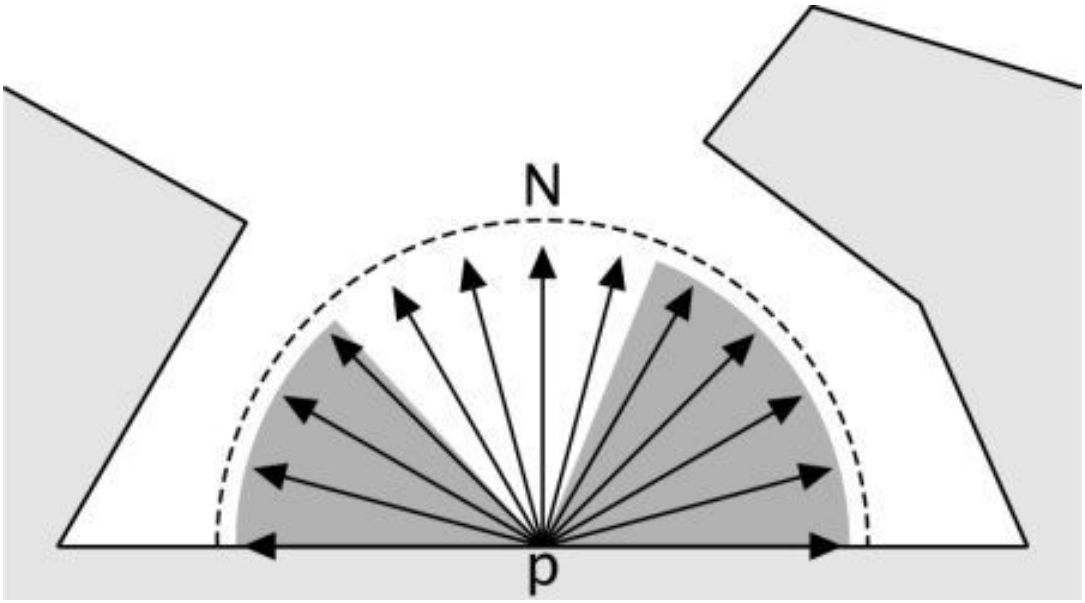

Figure 2.1: Ilustration of ambient occlusion, provided by Nicolas Bertoa in his blog post DIRECTX 12 – AMBIENT OCCLUSION – THE SECOND APPROACH

We can apply the notion of ambient occlusion to the reflectance equation to calculate the amount of light that leaves a point in the view direction, arriving at the following equation:

$$L_o(p,v) = \int_{l \in \Omega} f(l,v) L_i(p,l) v(p,l) (n \cdot l)^+ dl \tag{2.3}$$

where:

- $L_o(p,v)$ is a floating-point value, representing the light that leaves a point p in a view direction v
- Ω is a set of three-dimensional directions from which light could hit the surface
- l is a 3D vector, representing one of the directions in the Ω set
- $f(l,v)$ is a function which returns a floating-point value and represents the BRDF of the surface for the incoming light direction l and view direction v
- $v(p,l)$ is a function which returns a floating-point value and represents the visibility function of a position p in the direction l
- n is a 3D vector, representing the normal of the surface.

An alternative approach to defining ambient occlusion would be the fraction of the directions from which light could hit the surface, which are occluded. This can be done

through integrating the visibility function and weighting the values such that directions closer to the surface normals are more important, which is achieved through normalized cosine weighting of the integral. This approach leads to the following equation which gives the fraction of the surfaces hemisphere which is occluded, also known as the ambient occlusion factor k_A:

$$k_A(p) = \frac{1}{\pi} \int_{l \in \Omega} v(p,l)(n \cdot l)^+ dl \tag{2.4}$$

Unfortunately, this equation can only be used with BRDF's which are constant, due to them being able to be extracted from the reflectance equation's integral. As such, if only Lambertian surfaces need ambient occlusion or it is only intended to apply ambient occlusion to ambient lighting the ambient occlusion factor can be utilized, merely needing to multiply it by the Lambertian BRDF or ambient light respectively. However, if a more complex lighting setup is required, equation 2.3 must be used instead, which can be very expensive to evaluate.

The visibility function introduced by Landis [31] was based on ray casting and resorted to casting rays randomly throughout the hemisphere of the surface, checking if they hit an object or not. If they hit an object the visibility function would return 0, otherwise, it would return 1. Despite producing impressive results, this visibility function has problems. For one it fails for enclosed geometry. As an example, if an object is inside a house, that object will always have a visibility value of 0. The second problem being that it produces overly dark images due to not account for indirect lighting.

Zhukov et al. [63] introduced a solution for the enclosed geometry problem which they called obscurance (Figure 2.2). Obscurance substitutes the visibility function by a distance mapping function $\rho(p,l)$, allowing for the reformulation of equations 2.3 and 2.4 accounting for this change. The ambient occlusion factor equation would now have the formulation

$$k_A(p) = \frac{1}{\pi} \int_{l \in \Omega} \rho(p,l)(n \cdot l)^+ dl \tag{2.5}$$

The distance mapping function is nearly identical to the visibility fundamental, except for one key difference, it is capable of outputting values between 0 and 1 and not only 0 or 1. To achieve this, it tracks the distance traveled until the ray cast hits a surface, which is then divided by a preset value d_{min} which defines the minimum distance a ray must travel before the function returns the value of 1. We can then define the distance mapping function in terms of the distance traveled by a ray cast from a given point in a given

direction before that ray hits a surface, as follows:

$$\rho(p,l) = \frac{d_{traveled}(p,l)}{d_{min}} \tag{2.6}$$

Figure 2.2: Bungie utilizes precomputed ambient obscurance in its video-game Destiny

Ambient occlusion producing images darker than their real counterparts is caused by assuming that no light comes from blocked directions, however, in reality, indirect lighting causes light to come from those directions, these are called interreflections. While there are some theoretical approximations of this effect, if we want to model interreflections accurately we need to solve the recursive problem of the rendering equation[23], to shade one point we need to have already shaded another one. In practice, this is too expensive to be done in a real-time scenario.

2.1.2.2 Precomputed Ambient Occlusion

Computing the ambient occlusion factors for every surface can take a long time. For static scenes, this can be done ahead of time, before rendering.

Monte Carlo Methods are the most common way to precompute the ambient occlusion, where either the visibility function approach or the obscurance distance mapping function approach can be used. Rays are cast in random directions and their intersection with other objects is checked, and the ambient occlusion factor equation is evaluated numerically, through the following equation:

$$k_A(p) = \frac{1}{N}\sum_{i=1}^{N} v(p,l_i)(n \cdot l_i)^+ \tag{2.7}$$

where:

- $k_A(p)$ is a floating-point value, representing the ambient occlusion factor for position p
- N is an integer, representing the number of random-samples used for the Monte Carlo integration process
- $v(p, l_i)$ is a function which returns a floating-point value and represents the distance mapping function
- l_i is a 3D vector representing the i^{th} random direction to cast a ray in
- n is a 3D vector representing the surface normal

An approximation that can be used is substituting the cosine weight factor with importance sampling. Instead of casting rays evenly along the hemisphere of incoming light directions and utilizing the cosine weight factor, we imbue the cosine weight into the ray cast distribution. In other words, we cast rays in a way where they are more likely to be sent in the direction of the surface normal, as those results are likely to have a higher contribution to the overall occlusion. This approach is called Malleys Method.

This covers how ambient occlusion factors are precomputed, however just as important is how these are stored and used in real-time.Occlusion data is specific to every point of a surface, typically this data is stored in texture, volumes, or as part of the mesh vertex data.

Kontkanen and Laine[29] rely on the reciprocal of a quadratic polynomial to encode how ambient occlusion changes with distance from the object. The coefficients of the reciprocal are stored in a cube map, which they call an ambient occlusion field. This allows them to model the directional variation of occlusion and the ambient occlusion effect an object has on its surroundings. During runtime, coefficients are retrieved from the cube map and reconstructed, which is done utilizing the distance and relative position of the occluding object.

However, *Kontkanen and Laine's*[29] approach has one key issue, it requires reconstructing the ambient occlusion factor, which inherently incurs a performance penalty. *Malner et al.*[38] present a more straightforward and less computationally-intensive approach. They store ambient occlusion factors in a 3D texture which they call an ambient occlusion volume. Optionally, bent normals can also be stored alongside the ambient occlusion factors. With this approach, ambient occlusion factors just need to be read from the texture, there is no need for any computation, meaning this method has a lower runtime cost.

Kavan et al.[26] developed an approach based on signal processing to improve interpolated ambient occlusion factors, they called this method Least-Square Baking. They account for the fact ambient occlusion is a continuous signal, and that the calculation of ambient occlusion is a sampling process and that before rendering, data is interpolated, this being a reconstruction process. From this, they gleaned that tools from signal processing can be used to improve the quality of this sampling-reconstruction process. They sample the occlusion signal uniformly across the mesh and vertex values are derived in a way that minimizes the distance between samples and the interpolated data in the least-square sense. This method was developed to be used with the vertex data storage method, however, it can be applied to texture and volumes.

The methods presented here covered specific ways of storing precomputed ambient occlusion and utilizing precomputed ambient occlusion, however, there are other ways of doing this, which will be explored in further subsections. That being said going over all possible storage methods is beyond the scope of this work, and the methods present are commonly used for real-time precomputed ambient occlusion. In general, the storage methods utilized for diffuse and specular global illumination can also be used for this. Storage challenges and details of the methods are independent of the signal type.

2.1.2.3 Dynamic Object-Space Ambient Occlusion

Precomputing ambient occlusion is great for static scene elements, however, if we want to apply it to moving objects, or objects that change shapes, precomputing ambient occlusion is simply not an option. However, casting hundreds of rays per surface is also too computationally expensive to be used in practice, this means we need to rely on approximations.

Bunnel[4] presents an approximation where they model surfaces as collections of disk-shaped elements located on mesh vertices (Figure 2.3). This allows them to approximate both ambient occlusion and bent normals. Disks were chosen because, the occlusion one disk casts on another can be calculated analytically, removing the need for ray casting. That being said merely summing occlusion from all disks doest not work correctly, it would lead to an overly dark image due to disks behind other disks still counting towards occlusion, this issue is called double shadowing. To solve this Bunnel uses a two-pass approach, during the first pass they do a simple sum of all occlusion factors with double shadowing, during the second pass each disk's occlusion contribution is reduced by its occlusion factor from the previous pass. This method is capable of producing convincing results, however, it has a significant computational cost, computing all disk pairs in the scene has a $\mathcal{O}(n^2)$ runtime complexity, making it unusable in practice. Bunnel utilizes

a hierarchical tree to optimize computations against distant disks, which optimizes runtime complexity down to $\mathcal{O}(n \log n)$, making it much more manageable. *Hoberock*[18] improved upon this method's image quality, but at a performance cost tradeoff.

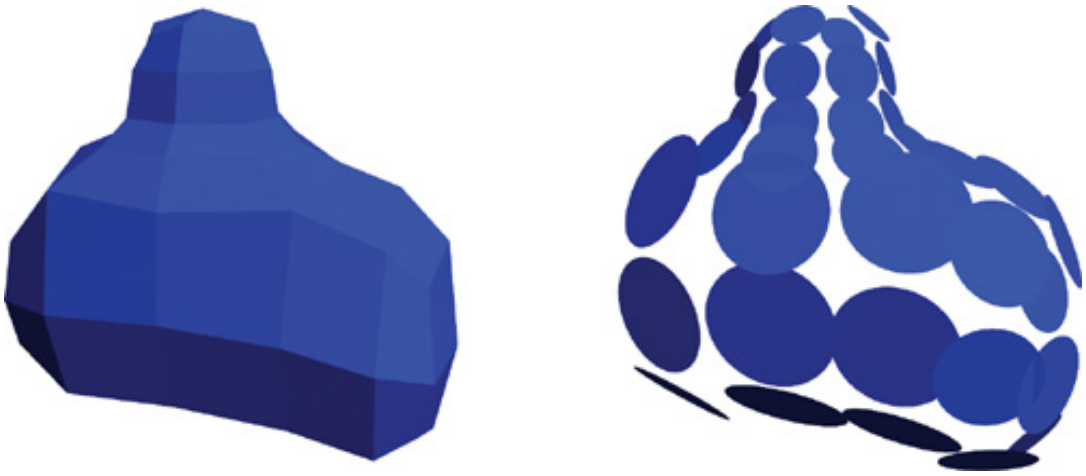

Figure 2.3: Example of the disk-based geometry approximation proposed by Bunnel, provided by Nvidia.

Evans[12] introduced a technique based on signed distance fields where objects are embedded into a 3D grid, where each grid position stores the distance to the nearest object surface. Negative values indicate the grid location is inside the object, and positive values indicate it's outside the object. Occlusion factor calculation is done via a heuristic, it combines values sampled at a given number of points of increasing distance along the surface normal. This method is capable of producing visually pleasing results. *Wright*[61] expanded on this approach, substituting *Evans'* ad hoc heuristic with cone tracing. Cones originate at the surface shading point and are tested for intersection against the scene's SDF representation. He approximated cone tracing by utilizing sphere of increasing radius to check for intersections, if an occluder's distance to the center of the sphere is under its radius then that surface is occluded. One of the issues with this approach is that several cone traces are required to achieve believable results, *Wright*[61] traces several cones that cover the surfaces hemisphere of incoming light direction to estimate ambient occlusion. To increase visual fidelity a scene global SDF is kept alongside the individual objects SDF's.

Crassin et al.[7] presented a method similar to *Wright's*[61], however, instead of representing the scene with signed distance fields, he utilized a voxel representation. Computing ambient occlusion through this method is just a special case of the full global illumination solution he presents. Nvidia has implemented this technique into its *VXGI - Voxel Accelerated Global Illumination* package, naming the ambient occlusion technique *VXAO - Voxel Accelerated Ambient Occlusion*.

Ren et al. [49] presents a method where they approximate the scene geometry through a collection of spheres. The reason for this is because the visibility function of a point occluded by a single sphere can be modeled with spherical harmonics. Additionally, the aggregate of all visibility functions can be quickly computed by summing the logarithm of all visibility functions and exponentiating the result. This computes not only ambient occlusion, but also, a full spherical visibility function, encoded using spherical harmonics. First-order coefficients contain the ambient occlusion factors, the next 3 coefficients contain bent normals and higher-order coefficients can be used for environment map shadowing for example. A downside of this approach is that since the geometry is approximated through spheres, occlusion from small details and creases are not accounted for with this model.

With the introduction of hardware-accelerated ray tracing and its API support through DirectX Ray Tracing and Vulkan Ray Tracing, we are capable of implementing, the original formulation of ambient occlusion, arriving at RTAO - Ray-traced ambient occlusion. RTAO allows us to trace rays in order to calculate visibility, instead of utilizing approximations. The distance mapping function presented by *Zhukov et al*[63] is the regarded as the best option due to producing more believable results, as well as being less expensive to compute due to the maximum distance bound. This being said, the number of ambient occlusion factors that need to be computed can be very high, leading to an inability to have large numbers of samples per surface when calculating ambient occlusion. In practice 1 to 2 samples per pixel/surface tend to be used, however, this leads to a very noisy image, to solve this, denoising algorithms are used. Currently, convolutional denoisers with spatial and temporal components are commonly used, however, some believe a deep learning approach may be feasible.

This category of dynamic methods has, however, one key issue in common, computational complexity is not constant as it scales with scene complexity, leading to different scenes possibly having very different performance costs.

2.1.2.4 Dynamic Screen-Space Ambient Occlusion

Unlike object-space methods, screen-space methods scale constantly with output resolution, meaning that independently of scene complexity, their performance cost will always be the same, only increasing when output resolution increases. To achieve this they rely only on screen-space data, mainly depth buffer and surface normal data.

Crytek[44] was the first to introduce a screen-space method aimed at solving the ambient occlusion problem. This method utilizes the Z-Buffer as its only input and computes ambient occlusion at full resolution. Ambient occlusion factors are computed for each

pixel by testing several random samples of points in the spherical neighborhood of the point against the Z-Buffer. Ambient occlusion factors are then a function of the number of samples that supersede their corresponding data in the Z-Buffer. Additionally, the importance of a sample diminishes with its distance from the pixel being tested. Unfortunately, since this method is not cosine weighted, it accounts for occlusion that should not be accounted for, leading to blackened edges around objects, sometimes referred to as the halo effect. That being said results are still visually pleasing.

Loos and Sloan[34] expanded upon *Crytek's approach*[44] and formulated it as a monte carlo integration, to which they gave the name Volumetric Obscurace and defined as:

$$v_A(p) = \int_{x \in X} \rho(d(p,x))O(x)dx \tag{2.8}$$

where X is a 3D spherical neighborhood around the point, $\rho(d(p,x))$ is a distance mapping function, and $O(x)$ is the occupancy function. They noticed that the distance mapping function had little impact on the visual quality, meaning it can be substituted for a constant function, simplifying the calculation. While *Crytek* computes the integral through random sampling, *Loos and Sloan* only compute through random sampling in the xy-dimension, integrating analytically in the z-dimension. In consequence, their method requires a lower amount of samples to achieve the same image quality as *Crytek's*.

Basing itself on *Max's Horizon Mapping technique*[40], *Bavoil et al.*[2], introduced a technique which they called HBAO - Horizon Based Ambient Occlusion. This method interprets the Z-Buffer data as a continuous heightfield, computing visibility at a given point by determining its horizon angles. For a given direction from a point, the horizon angle is the angle of the highest visible object from that point. This method is then defined as:

$$k_A = 1 - \frac{1}{2\pi} \int_{-\pi}^{\pi} (sin(h(\phi)) - sin(t(\phi)))W(\phi)d\phi \tag{2.9}$$

where, $h(\phi)$ is the horizon angle above the tangent plane for the given direction, $t(\phi)$ is the tangent angle between the tangent plane and the view vector, and $W(\phi)$ is the linear falloff attenuation function. This integral can be computed numerically through random sampling.

Continuing with the idea of horizon mapping, *Jimenez et al.*[22] introduced a technique which they called GTAO - Ground Truth Ambient Occlusion. It aims to achieve ground-truth results, equaling those produced by ray-tracing. Just like the methods before it relies only on Z-Buffer data, and interprets it as a heightfield. Unlike HBAO, it

introduces cosine weighting into the ambient occlusion factor computation and removes HBAO's ad-hoc attenuation function. The occlusion integral is formulated in the reference frame around the view vector as:

$$k_A = \frac{1}{\pi}\int_0^{\pi}\int_{h_1(\phi)}^{h_2(\phi)} cos(\theta-\gamma)^+ |sin(\theta)| d\theta d\phi \tag{2.10}$$

where $h_1(\phi)$ and $h_2(\phi)$ are the left and right horizon angles respectively, and γ is the angle between the surface normal and view direction. The inner integral can be solved analytically while the outer integral must be integrated numerically, this is done through random sampling directions around the pixel. The most expensive aspect of this approach is sampling the depth buffer along screen-space lines, *Timonen*[56] presents an approach to mitigating the impact of this step.

Despite providing a much more consistent and predictable performance cost, screen-space approaches present several drawbacks:

- Most methods will present a halo effect, where they create a contour around objects. (Figure 2.4)
- The depth buffer is not a perfect scene representation, leading to the usage of heuristics that attempt to extract more information out the data set.
- A large number of samples of the Z-Buffer will be required to achieve convincing results.
- Dithering tends to be required to bridge the gap between practice and theory.

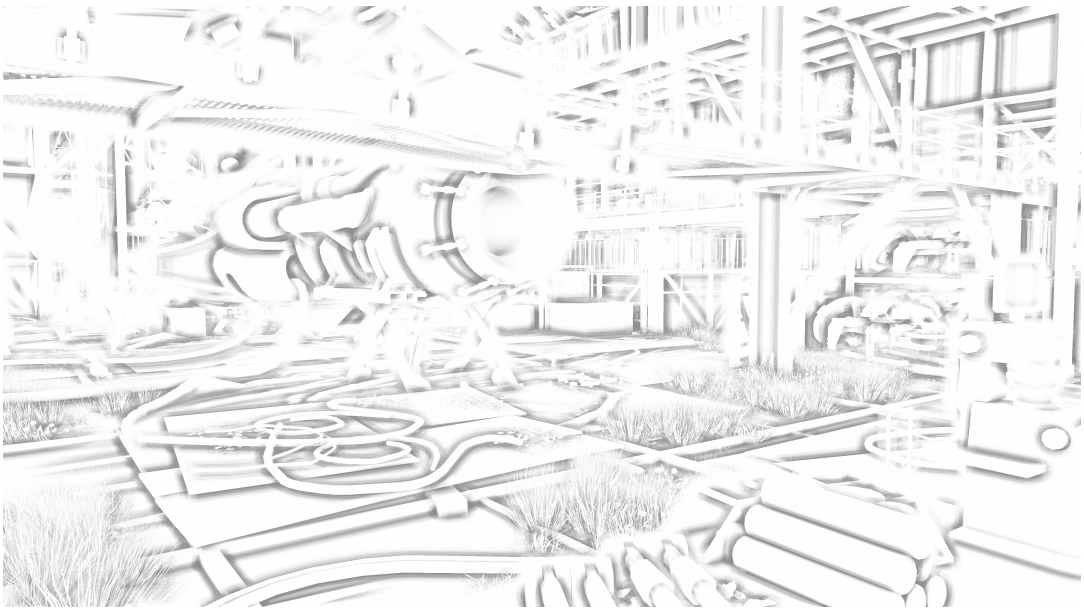

Figure 2.4: Outline around object caused by screen-space ambient occlusion in Destiny 2, provided by Nvidia.

2.1.2.5 The Issues With Ambient Occlusion

Despite producing highly believable results at a relatively low performance cost, ambient occlusion still represents a massive oversimplification and approximation of global illumination. It cannot handle small area lights or punctual lights, due to how easy they are to occlude. It cannot handle complex lighting setups, such as when a light is partially occluded. Lastly, its performance cost for any surface that is not Lambertian is significantly higher, and prohibitive in a real-time setting.

2.1.3 Directional Occlusion

As explained in section 2.1.2, ambient occlusion is significant, simplification of global illumination, utilizing a very poor approximation of visibility as well as not being able to handle glossy BRDFs or complex light setups. Directional occlusion aims to solve these issues by describing visibility in a more accurate form.

Methods presented here, can not only be used for encoding visibility but also to provide shadowing in situations where traditional techniques are not viable, this is the case for self-shadowing of bump map details and for shadowing extremely large scenes.

2.1.3.1 Directional Occlusion Theory

Directional occlusion methods encode the complete spherical or hemispherical visibility and look for a way of describing which directions are blocking incoming light. Methods for directional occlusion mainly focus on large area-lights or environment mapping, this is because they can assume that the generated light is soft and that artifacts generated by approximations are easy to notice.

Just like in ambient occlusion we start with the reflectance equation, extended to include a visibility check:

$$L_o(p,v) = \int_{l\in\Omega} f(l,v)L_i(p,l)v(p,l)(n\cdot l)^+ dl \tag{2.11}$$

However unlike in ambient occlusion unrolling this integral will depend on more factors, how we encode visibilty, surface type and the type of light being used. Depending on the type of light unrolling this integration can become quite simple such as in the case of point ligths, or extremely complex such as with area lights. Just like with ambient occlusion if we are dealing with a Lambertian surface we can extract its BRDF from the

integral, however, that will not be the case for other surfaces, so despite glossy surfaces being supported, they are prohibitively expensive to compute in real-time in most scenarios.

There has been a great deal of effort put into approximating aspects of this computation, however, the subject is very deep due to how many aspects the equation depends on. For this reason, a thorough analysis of this subject out of the scope of this work.

2.1.3.2 Precomputed Directional Occlusion

Precomputing ambient occlusion involves both determining visibility and encoding it in an efficient format. Unlike ambient occlusion there is not well-known and established approach to computing visibility, meaning precomputed directional occlusion methods will have a higher degree of variance compared to ambient occlusion precomputed methods.

Max[40] developed the horizon mapping approach to describe the self-occlusion of a heightfield. Visibility is calculated by determining the altitude or horizon angle for a set of azimuth directions for every point in the surface. This data is then stored into a horizon map.

Oat and Sander[46] developed a technique which they call ambient aperture lighting (Figure 2.5). They approximate the shape of unoccluded regions above the shading point via a cone, occlusion of an area-light can then be estimated by intersecting the light-cone with the occlusion cone. Then the set of 3D unoccluded directions is modeled as a circular aperture. This method presents a lower storage requirement than horizon maps. The main drawback of this approach is that if the set of unoccluded directions does not resemble an ellipse or a circle, it can produce incorrect shadowing results.

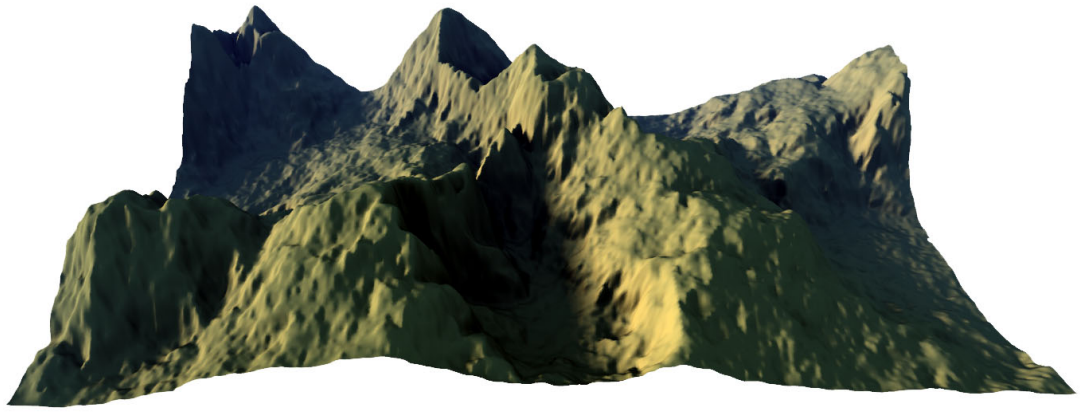

Figure 2.5: Image rendered with ambient aperture lighting, provided by authors.

Wang et al.[57] represent visibility in a structure called a spherical signed distance field, it encodes the signed distance to the boundary of the occluded region in the sphere.

Other spherical or hemispherical based encoding methods can be used to encode visibility. Just like ambient occlusion, directional occlusion can be stored in textures, mesh vertices or volumes.

2.1.3.3 Dynamic Directional Occlusion

Some methods developed for ambient occlusion or also able to provide directional occlusion information or can be extended or modified to provide this information. *Ren et al.'s spherical harmonical exponentiation*[49] encoded visibility in the form of spherical harmonic vectors, therefore if more than one spherical harmonic band is used, it will provide directional occlusion. In general, the more spherical harmonic bands used, the more precise the visibility encoding.

Crassin et al.'s[7] voxel cone tracing approach provides an occlusion value for each cone trace, and since we already need to trace multiple cones to achieve acceptable image quality, it already produces directional occlusion data. If visibility is only required in a single direction, a smaller number of cones can be traced. *Iwanicki*[20] utilize cone tracing, restricting it to a single direction and storing lighting for static geometry using the AHD encoding. This results in soft shadows cast on static geometry by dynamic objects which are approximated by spheres. The visibility for the ambient and directional occlusion components can be handled independently. Ambient occlusion is computed analytically, directional occlusion is computed by tracing a single cone and intersecting it with the sphere object representation.

Klehm et al.[28] present a screen space method to compute directional occlusion. They rely on Z-Buffer data to compute screen-space bent cones, which are circular apertures like those precomputed by *Oat and Sander*[46]. When they sample the neighborhood of a pixel, they sum all the unoccluded direction vectors, the length of the results vector can be used to estimate the apex angle of the visibility cone, the direction of the result vector determines the axis of the visibility cone. *Jimenez et al.*[22] utilizes horizon angles to compute the axis of the visibility cones and derives the apex angle based on the ambient occlusion factor.

2.1.3.4 Issues with Directional Occlusion

Directional occlusion solves some of the issues of ambient occlusion, producing a noticeable image quality improvement, however, it is not a perfect solution, many of ambient occlusion's problems are still present:

- It still does not handle well small area-lights or punctual lights.
- Interreflections are not accounted for, leading to darker than reality scenes.
- The performance cost is significantly increased for all surfaces; compared to ambient occlusion; varying with visibility encoding, light type, and surface type.

2.1.4 Diffuse Global Illumination

Diffuse global illumination aims to simulate not only occlusion but also how light bounces around the scenes and reaches the eye. In order to achieve global illumination in real-time, the field relies on specific assumptions and approximations. For diffuse surfaces, it is assumed that incoming light either changes smoothly across the hemisphere of the shading point, or it can be ignored altogether.

Just like ambient occlusion and directional occlusion, we can broadly categorize global illumination methods into those that are precomputed and those that are dynamic. There are essentially two categories of precomputed diffuse global illumination methods, those that rely on directional surface prelighting and those that rely on precomputed radiance transfer. Dynamic methods, on the other hand, are much more varied and therefore harder to subcategorize.

2.1.4.1 Directional Surface Prelighting

Radiosity and path tracing can be used to precompute diffuse global illumination, the question then becomes how to encode and store them in a way that is usable in a real-time environment. This precomputing option is sometimes referred to as full baking, this is derived from the fact that any change in the scene would invalidate the precomputed lighting, meaning nothing in the scene can change for the lighting to remain accurate. Despite being a significant compromise it is an acceptable one in some scenarios. The simplest type of lighting that can be precomputed is irradiance, and because the effect from light sources is independent of each other, dynamic lights can be utilized on top of precomputed irradiance.

Traditional precomputed irradiance, however, has some restrictions, due to it being computed for a given normal distribution, normal mapping cannot be utilized, on top of that irradiance can only be precomputed for flat surfaces. To solve this, it is required to model how irradiance changes with the surface normal. To provide indirect lighting for dynamic objects we need to know the lighting value for every possible surface orientation.

Good and Taylor[14] present an approach where they encode full spherical irradiance information. It is the most generic method to encode directional surface lighting and

achieves this through spherical harmonics. Spherical harmonic coefficients are stored in textures, with 3^{rd} order spherical harmonics (9 coefficients), the resulting quality is very high, however, it has much more significant storage and bandwidth cost. A possible tradeoff is dropping down to 2^{nd} order spherical harmonics (4 coefficients), however, this leads to a loss of many subtleties, a loss in lighting contrast and normals maps become less pronounced, however, it does come at a significantly lower cost. *Chen and Liu*[5] present a variation of this method, which they utilized in Halo 3. This technique aims to achieve the quality level of a 3^{rd} order spherical harmonic at a much lower cost. They extract the most dominant light and store it separately as color and direction, residual data is then encoded in 2^{nd} order spherical harmonics. This results in very little quality loss at a much lower cost.

Habel and Wimmer[15] utilize a basis they call H-Basis to encode a hemispherical signal. With the basis they present, fewer coefficients are required to achieve the same quality as with 3^{rd} order spherical harmonics. However, a coordinate system that is local to the surface is required to orient the hemisphere. For this purpose, a common approach is utilizing a tangent frame resulting from uv-parameterization. In this case, if the H-Basis is encoded in textures, this texture needs to have sufficient resolution to adapt to changes in the underlying tangent space. However, has one key issue, if two triangles cover the same texel and have a significantly different tangent will lead to a lack of precision in the reconstructed signal.

Both the spherical harmonics and H-Basis approaches have a couple of issues in common. One is that they both can exhibit ringing, prefiltering can mitigate this effect, however, it also leads to smoothening of the lighting, which is undesirable. The other issue is that they both present relatively high memory and bandwidth costs.

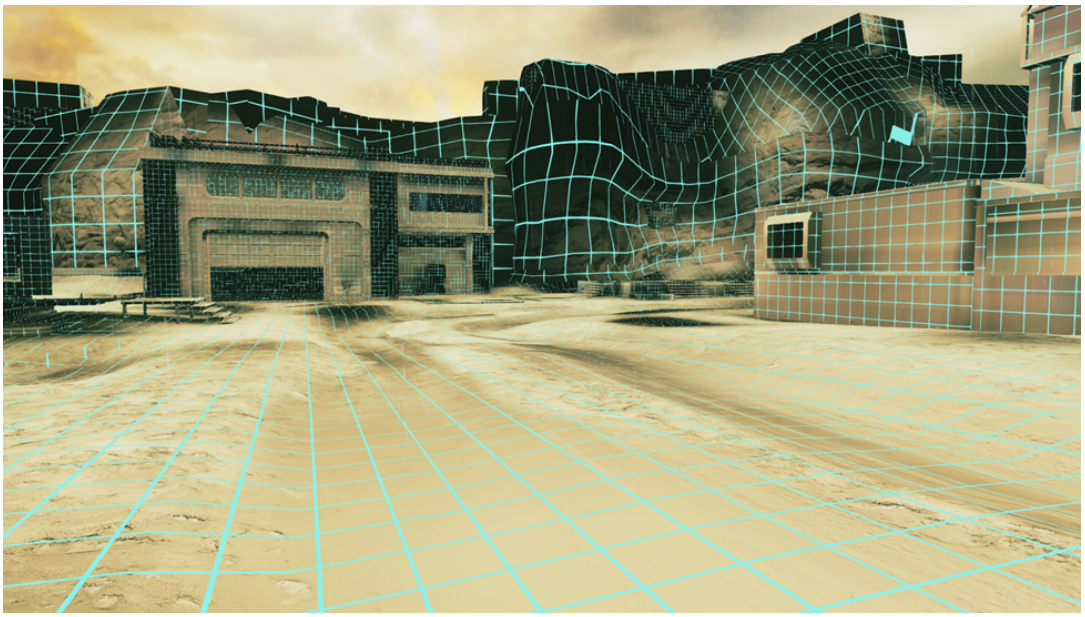

Figure 2.6: Baked lightmaps in Call Of Duty: Infinite Warfare using AHD encoding, provided by Activision.

Neubelt and Pettineo[45] developed an approach for the video game The Order: 1886, that focuses primarily on image quality. They store coefficients in texture maps, utilizing spherical gaussian's to encode lighting. These coefficients store incoming radiance instead of irradiance, they are projected to a set of gaussian lobes in a tangent frame, 4-9 lobes are used depending on the complexity of the lighting in the scene. To produce a diffuse response from the material, spherical gaussian's are convolved with a cosine lobe oriented along the surface normal. This representation is even precise enough to encode low-gloss specular effects by convolving gaussian's with the specular BRDF lobes.

2.1.4.2 Precomoputed Radiance Transfer

Directional surface prelighting is inherently static, and, while this is a fine tradeoff in certain situations, it is very inflexible, to the point where it is unusable for certain scenarios. To obtain a greater degree of flexibility we aim to find a way to model how light interacts with surfaces without operating on actual radiance values. Transfer functions allow us to do this, they allow us to turn incoming light into a description of how radiance is distributed throughout a scene. Precomputed solutions to this problem are called precomputed radiance transfer approaches, incidentally, they also have a higher runtime cost than full baking as we will need to calculate radiance values for a particular lighting setup at runtime.

Sloan et al.[51] introduced the concept of precomputed radiance transfer to computer graphics. While they formulated it in terms of spherical harmonics, it does not require the

usage of spherical harmonics. They noted that the contribution of each light to the overall lighting of the scene can be computed independently and summed up to achieve the final lighting value, this is possible because light transport is linear. They formulated this as:

$$L(p) = \sum_i L_i(p) W_i \tag{2.12}$$

where $L(p)$ is the final radiance for point p, $L_i(p)$ is the precomputed unit contribution of light i and W_i is the current brightness of the light i. In a mathematical sense, this equation defines a vector space with L_i being the basis of this vector space. Sloan et al. store the scenes response to light with a distribution defined by a spherical harmonic basis function. They model the response for a number os spherical harmonic bands which allows them to render a scene with any arbitrary lighting setup. An important factor to consider is that the basis function is independent of the representation utilized to represent the final lighting. They later, went on to describe ways os reduce the memory cost of this approach.

Lehtinen et al.[32] describe a volumetric approach to calculate precomputed transfer. Their approach allows for the querying on any location in 3D space and provides consistent lighting between static and dynamic geometry. The main drawback of their approach is its runtime cost.

Mitchel et al.[43] describe a method where they utilize modular cells to precompute transfer. Multiple of these unit cells are stitched together and warped to approximate scene geometry. Radiance is then propagated first to cell borders which act as interfaces and then to the neighboring cells using precomputed models.

2.1.4.3 Dynamic Diffuse Global Illumination

Dynamic approaches to computed diffuse global illumination either don't require any preprocessing, or this preprocessing is fast enough that it can be computed every frame.

Keller[27] introduced instant radiosity which is one of the first approaches presented to approximate diffuse global illumination, despite the name, it has little in common with the Radiosity algorithm. In this method, rays are cast from a light source, when these rays hit a surface, a new light is placed there, effectively this simulated diffuse single bounce lighting. These new lights were called virtual point lights.

Tabellion and Lamorlette[53] present an approach they utilized in the production of Shrek 2. They perform a direct lighting pass of the scene and store the result in textures, then during final rendering, the method traces rays and uses the cached lighting to create

one bounce indirect lighting. This inspired *Dachsbacher and Stamminger*[8] in the creation of the reflective shadow maps technique. These just like normal shadow maps are rendered from the point of view of the light source. They store depth information and about surfaces, such as their albedo, normal and direct illumination. When performing the final shading pass, texels of the reflective shadow map are treated as point lights to provide single bounce indirect illumination. They optimize this process by reversing the processing, several lights are created based on the entire map and then they are splated in screen-space. This approach, however, has two key issues, if all texels of the reflective shadow maps are used the performance cost will be too great, leading to the need to use heuristics that chose regions of the map to be used at a given time. The other issue is that if the number of lights created is not enough this will lead to temporal stability issues when moving lights, results in flickering.

Kaplanyan[24] introduced a technique called light propagation volumes that also aims to simulate first bounce indirect illumination. In this approach the scene is discretized into 3D cells in a continuous grid, each cell will hold a directional distribution of radiance flowing through it, 2^{nd} order spherical harmonics are used to encode this data. In the first step of the computation, lighting is injected into the cells that contain directly lit surfaces, to find these cell reflective shadow maps are utilized, however, any other method can be utilized for this purpose. Injected lighting is then radiance reflected off the lit surfaces, leading to the formation of a distribution around the normal, facing away from the surface and who's color is based on the surface material. In the second step, light is propagated from cell to cell, each cell analyzes the radiance field of its neighbors and modifies its own based radiance coming from all directions. The number of steps allowed will control the light propagation. *Kaplanyan et al.*[25] developed a cascade variant of the original light propagation volume formulation presented by Kaplanyan.

In his Ph.D. thesis *Crassin*[6] introduced the now very popular approach called voxel cone tracing global illumination also called VXGI by Nvidia (Figure 2.7). This approach is based on a voxelized scene representation where geometry is stored as a sparse voxel octree. The reason for this is because a sparse octree provides a mimpap-like representation of the scene and a volume of space can rapidly be tested for various aspects, occlusion being an example. The voxels contain information about the amount of light reflected off the geometry they approximate, this data is stored in a directional format as the radiance is reflected in 6 major directions. Using reflective shadow maps, direct lighting is injected into the lowest levels of octree and propagated up the hierarchy, the octree is then used to estimate incident radiance. Ideally for this rays would be traced, however, this would require many rays and is therefore approximate through cone tracing. Cone intersections are approximated through octree lookups. While this method produced highly realistic

results, a naive implementation can lead to severe performance issues as a sparse octree does not map easily to the GPU programming model, leading to possible WARP stalling and therefore degrading performance.

Figure 2.7: Image rendered with voxel cone tracing global illumination, provided by Nvidia.

McGuire et al.[37] present a technique that blends hardware-accelerated ray tracing and traditional rasterization approaches into a method they call dynamic diffuse global illumination (DDGI) (Figure 2.8). Based on prior work from *McGuire et al.*[41] they extend classic irradiance probes to compactly encode the full irradiance field in a scene. One of their main concerns was to eliminate the light and shadow leak issues of other global illumination approaches. Hardware-accelerated ray-tracing based shading is blended with filtered irradiance queries to compute indirect illumination. Each frame updated ray-traced illumination is blended into the probe atlas and interpolated probe depth information, such that probe data adapts to changes in scene geometry. Probes are updated in the following steps:

1. n rays are generated and traced from each of the m active probes. Geometry hits are stored for up to m by n surfaces in a G-Buffer structure with position and normal data, called a surfel.

2. Surfels are shaded with direct and indirect illumination.

3. Texels in active probes are updated by blending in updated shading, distance and square-distance results for each of the intersected surfels.

in the method, they do not offer any heuristics for choosing which probes are active and which are not, meaning they are all active, however it is perfectly possible to utilize a heuristic to reduce the number of probes that need to be updated.

Figure 2.8: Image rendered with DDGI, provided by the authors.

2.1.5 Specular Global Illumination

This subset of global illumination focusses on simulating view-dependent effects. For glossy materials, specular lobes are much tighter than the cosine lobes used in diffuse lighting and for highly shiny materials, a radiance representation capable of delivering high-frequency details is required. A key assumption in this subset is that when evaluating the reflectance equation, we only need to consider radiance incoming from a limited solid angle. Methods that store incident radiance can be used to deliver coarse view-dependent effects. Based on these assumptions, *Xu et al.*[62] developed an approximation of the specular response of a typical microfacet BRDF, defined as:

$$L_o(v) = \sum_k [M(l_k, v)(n \cdot l_k)^+ + \int_{l \in \Omega} D(l, v) L_k(l) dl] \tag{2.13}$$

where $M(L_k, v)$ is the factor combining the fresnel and mask-shadowing functions, $D(l, v)$ is the normal distribution function; which Xu et al. model through an anisotropic spherical Gaussian, and $L_k(l)$ is the k^{th} spherical gaussian representing incident radiance. The primary issue with these approaches is that they are not precise enough even when using 3^{rd} order spherical gaussians, requiring a much higher number of coefficients, making them therefore unusable in a real-time scenario.

2.1.5.1 Localized Environment Maps

Localized environment maps are a popular approach to solving the issue with the precomputed approach outlined earlier. Incoming radiance is represented through an environment map, therefore few values are required to evaluate radiance. These tend to be sparsely distributed throughout the scene, this results in a loss of spatial precision of incident radiance, in favor of an increased angular resolution. If these environment maps are rendered from a specific point of view in the scene, they are commonly called reflection probes. If an object is small and the environment map is rendered from its center, the results produced will be highly accurate, however, this situation is rare. In general the further away the object is from the environment maps center, the more the resulting reflections will deviate from reality.

Brennan[3] present a solution to this issue, he assumes that incident lighting comes from a sphere with a finite size. Then, when accessing the environment map, he doesn't directly use the incoming radiance direction, he treats it as a ray originating from the surface location and intersects it with the light sphere. Then a direction is calculated from the environment map center to the intersection point, the is now the look direction. What this does is it fixes the environment map in space, this is commonly referred to as parallax correction.

We can extend reflection probes to other shapes, leading to the concept of reflection proxies, where any shape can be used as long as it can be intersected with a ray. The more the proxies shape and size match with the surface's, the more accurate the resulting reflections will be. Environment maps are traditionally precomputed, however, we can update during runtime at whatever rate we desire, such that they become dynamic and reflect what is happening in the scene. In a sense, this approach can either be classified as a precomputed or dynamic approach depending on design decisions.

2.1.5.2 Voxel Cone Tracing Global Illumination

Localized environment maps leave something to be desired out of the final image quality due to insufficient spatial density and crude shape approximations. *Crassin's*[6] voxel cone tracing approach presented in section 2.1.4 can also be utilized for specular global illumination. For glossy materials only a single cone trace is necessary, this leads to a lowered performance cost, compared to when it's used for diffuse global illumination. However, this approach is not perfect, when dealing with highly polished reflectors, the underlying voxel representation of the scene might be revealed, however, this rarely happens in practice.

2.1.5.3 Planar Reflections

Planar reflections are another alternative to localized environment maps. Planar reflections, reuse the regular scene representation, opting to merely rerender it from a different perspective. We then store the result of this render into a texture that we will use in the final render pass. The main issue with this approach is the requirement of a render pass for each mirror surface, this becomes even more of an issue when dealing with recursive reflections. This issue can make planar reflections prohibitively costly.

2.1.5.4 Screen-Space Reflections

Screen-space reflections commonly abbreviated to SSR and first presented by *Sousa*[52], are very commonly found in real-time scenarios to render reflections. Given a point being shaded, the view vector and the surface normal, if we reflect the view vector along the surface normal, we can trace a ray originating from the surface with the reflected view vector as its direction, we then test this ray for intersection with the depth buffer. This is done by iterating over the ray, computing its screen-space coordinates and fetching the corresponding Z-Buffer data, then, if the point is further away from the camera than the geometry in the depth buffer, that means that the ray is inside the geometry. Then we can read the color buffer to obtain the incident radiance value from the traced direction and utilize it to shade the surface. Up to this point the ray intersection location would be a somewhat imprecise value, this is because we don't know where in the distance traveled between ray steps the intersection occurred, we can utilize binary search between the last two steps to improve the intersection point estimation. The two issues with this approach are that it will only work on Lambertian surfaces and that it can only reflect what is on screen.

2.1.5.5 Real-Time Specular Reflections With Radiance Caching

Hirvonen et al.[17] present an approach to specular global illumination that blends hardware-accelerated ray tracing with traditional rasterization techniques. They enhance probe and screen-space based techniques with ray tracing, presenting a way to combine the techniques, a heuristic for probe sampling as well as an alteration to motion vectors that provides temporal reflection filtering. This approach presents 4 distinct stages:

1. Radiance cache creation for static geometry
2. Lighting of radiance cache
3. Radiance sample generation

4. Reflection filtering

an important aspect to note is that radiance caches will only include static geometry. The radiance cache creation step is treated as a preprocessing step for static geometry (Figure 2.9). Radiance cache lighting is decoupled for reflection ray tracing and the sampling pass, ray not found in the radiance cache are shaded with a normal ray tracer after this, results are filtered with a spatiotemporal filter. The radiance cache is a radiance probe that stores a cube map, the data of this probe is filtered in screen space and probes can either be placed manually or automatically, however, the authors do not offer an approach to automatic probe placement. Probes are reilluminated every frame and ray-tracing is used to generate sampling directions according to the specular BRDF in use. In short, rays are traced from the camera with a single bounce, when they hit a surface if that surface is visible from a probe data is fetched from the probe for shading, otherwise, regular ray-tracing based shading occurs (Figure 2.10).

Figure 2.9: Image rendered using specular reflections with radiance caching, provided by the author.

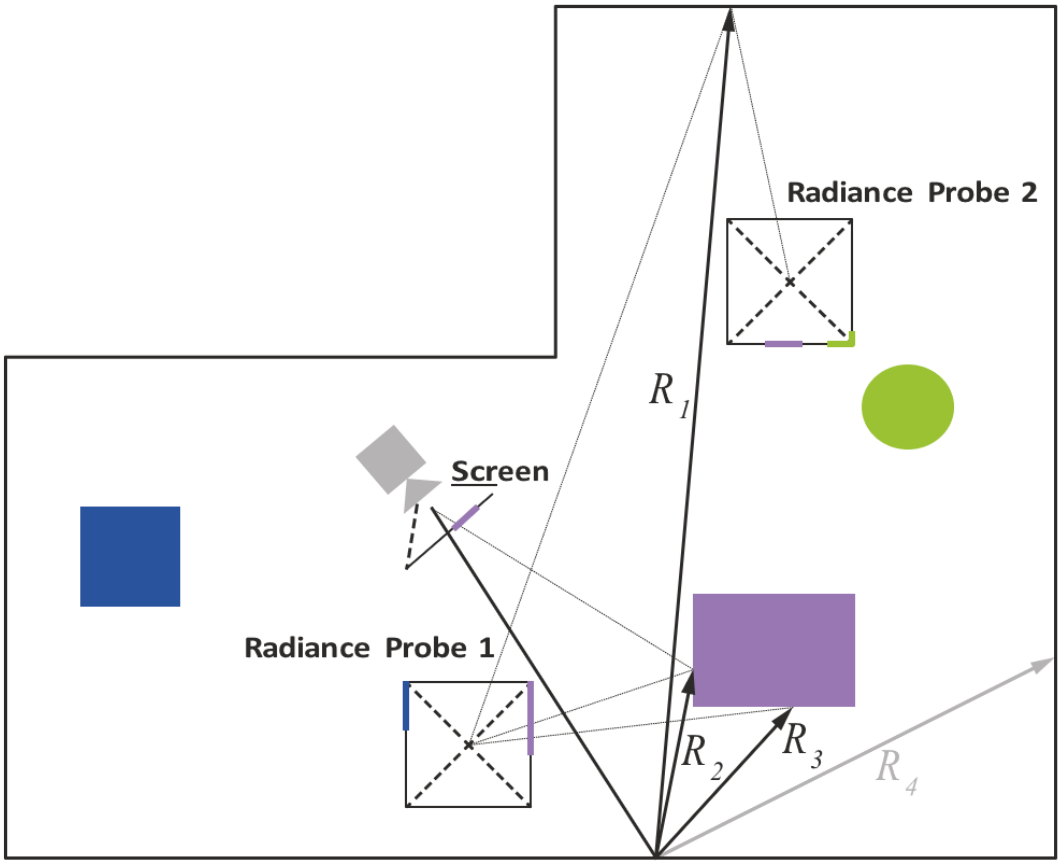

Figure 2.10: Diagram demonstrating basic integration of reflection probes and ray tracing, provided by the author.

2.2 Image Quality

Image quality analysis if a fundamental part of understanding whether or not a computer graphics method meets its objectives and accurately simulates reality. It is then fundamental to incorporate image quality analysis into the process of validation of computer graphics methods. This chapter presents approaches to analyze the perceived quality of an image. First, it goes over how humans perceive images, and then it goes over approaches to assessing image quality.

2.2.1 The Human Visual System

The psychophysics of human visual perception, indicate that the features of the human visual system do not activate separately but rather in tandem to produce the perception of a given image. *Palmer*[47] presents a comprehensive description of the characteristics

of the human visual system. An important concept to understand is that of visual acuity, visual acuity refers to the ability of the human visual system to resolve detail in an image.

Some of the most important characteristics of the human visual system are the following:

- The human eye is less sensitive to gradual and sudden changes in brightness in an image but has a much higher sensitivity to intermediate changes.
- The human visual system has a higher monochromatic spatial acuity than chromatic spatial acuity.
- Under low levels of illumination the human eye is very sensitive to changes in luminance, however, it is much more difficult to discern shape and color. In contrast, under high levels of illumination, we can easily perceive shape and color, however, changes in luminance need to be much more drastic in order to be perceived.
- Human perception is more sensitive to lower spatial frequency than higher ones, if the spatial frequency is too high we will seise to perceive changes entirely, this is expressed through the contrast sensitivity function (CSF). The contrast sensitivity function tells us how sensitive the human visual system is to various frequencies of visual stimuli. Contrast sensitivity depends therefore on spatial frequency.
- Human eyes are sensitive to luminance contrast and not absolute luminance levels, this means that apparent brightness will depend on the background, this phenomenon is called simultaneous contrast (Figure 2.11). This happens because human brightness sensitivity is a logarithmic function.
- The human visual system presents a phenomenon called contrast masking, where visibility of a pattern can be reduced by the presence of another pattern.

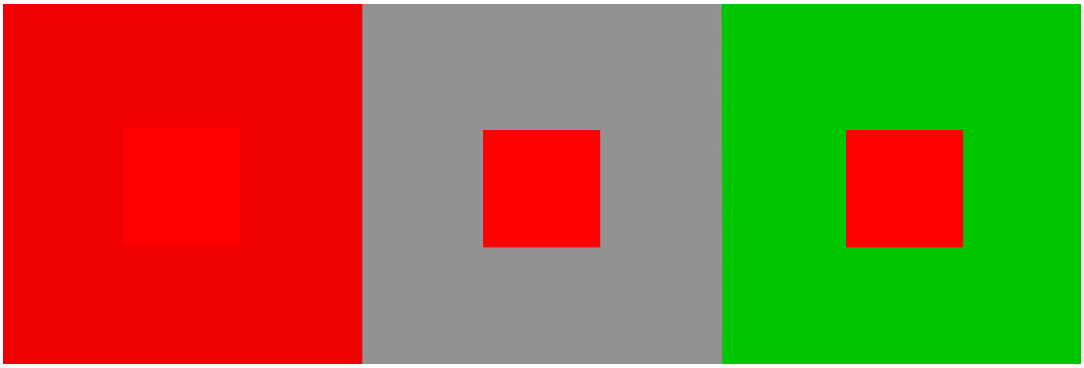

Figure 2.11: Example of simultaneous contrast, the central square always has the same color and brightness, however its perception depends on the background square.

The list presented here merely encompasses aspects that affect our image perception, however, the human visual system is much more complex, that being said a full exploration of it lies far beyond the scope of this work.

2.2.2 Image Quality Analysis

Analyzing image quality is usually done through a comparison of two or more images. This section presents approaches to analyzing image quality, be them automatic or through experimental means with human subjects.

2.2.2.1 Automatic Image Quality Perception Analysis

Automatic image quality measurement can be classified regarding the availability of a reference image and regarding the general approach they take to compute the image quality measurement. Regarding the availability of reference images, we can have a full reference(FR), a reduced reference (RF) or no reference at all. Regarding approach most methods fall under the objective mathematical approach or under the approach that incorporates elements of the human visual system, however, some methods can fall outside the umbrella of these two.

The methods presented below are full-reference methods since these produce the best results, and given this work's needs, they are the best fit.

Mathematical Metrics These metrics treat an image as a 2D signal, they then calculate the dissimilarities between the reference image and the test image Minkowski's metric calculates the distance between a reference image x and a test image y, this is defined as:

$$E_\gamma = \left(\frac{1}{N} \sum_{i=1}^{N} |x_i - y_i|^\gamma \right)^{\frac{1}{\gamma}} \tag{2.14}$$

where N is the number of samples in the image, x_i and y_i are the i^{th} samples of image x and y and γ is a value in the range between [1,∞[. When γ has the value 2 we arrive at the equation for the Mean square error (MSE) if we ignore the square root:

$$E_2 = \left(\frac{1}{N} \sum_{i=1}^{N} |x_i - y_i|^2 \right)^{\frac{1}{2}} = \sqrt{MSE} \tag{2.15}$$

The mean square error metric is the most commonly used, however, *Eskicioglu*[11] presents a list with other mathematical distortion measures. From these, some highlighted approaches are the peak signal-to-noise ratio (PSNR), peak mean square error and the laplacian mean square error. While these mathematical metrics are highly objective, they correlate very poorly to how humans perceive images, as demonstrated by *Girod*[13]. The primary reason for this is due to them not accounting for the human visual system's characteristics in their models.

HVS-based Metrics *Janssen*[21] proposed a new philosophy regarding image quality where he treats images as carriers of visual information and not as signals, he regards visual-cognitive processing as information processing rather than signal processing and he regards image quality as image adequacy in the visual interaction process instead of visibility of distortions. Human visual system based metrics, therefore, take the difference between the reference and test image and normalize it according to its visibility by the psychophysics of human perception. Some notable HVS-based image quality metrics are:

- *Mannos and Sakrison's*[39] visual fidelity criterion
- *Daly's*[9] Visible difference predictor
- *Lubin's*[35] model
- *Watson's*[60] DCT-based and wavelet-based metric
- *Teo and Heeger's*[55] perceptual image distortion
- *Sarnoff's*[36] visual discrimination model

In general, these metrics map very well to a real user's image perception. The main downside of these metrics is their complexity, the nonlinearity of the human visual system and the suprathreshold problem described by *Wang et al.*[59]

Wang et al.'s structural similarity approach *Wang et al.*[59, 58] proposed a new approach to designing image quality metrics, which falls somewhat outside of the umbrella of the two categories previously outlined. They assume that the human visual system is highly adapted to the highly structured natural scene information. Therefore they measure how the structural information changes to provide a good approximation of perceived image distortion. Structural similarity is defined as:

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{2.16}$$

where μ_x and μ_y are the mean intensity of image block x and y, σ_x and σ_y are the standard deviation of image block x and y, σ_{xy} is the cross-correlation of the two image blocks and C_1 and C_2 are constants utilized to avoid instability problems.

2.2.2.2 Experimental Image Quality Perception Analysis

While automatic image quality assessment is great in situations where it's not feasible or practical to conduct user testing, it can only go so far, even the most advanced approaches will only ever approximate what a real user perceives. This means that user testing will always hold great value.

This being said experimental approaches to image quality assessment have their issues, such as controlling all variables that possibly alter the way a user perceives an image or achieving statistical validity. The ITU[19] provides a guide to conducting image quality assessment experiments on humans. The fundamental way in which these experiments are conducted is a choice between double stimulus and single stimulus.

With double stimulus, we present the reference image and the test image and ask the test subject to evaluate them on a linear scale, for this, we can use either the double stimulus continuous quality scale (DSCQS) or the double stimulus impairment scale (DSIS). With single stimulus, we only present the test image and ask the test subject to evaluate it on a linear scale. Scores must be evaluated by multiple subjects and then averaged for each image, however, we must first execute subject and outlier rejection. We can then obtain the mean opinion score (MOS) or the difference mean opinion score (DMOS) to conclude which image is perceived as having higher image quality.

Chapter 3

Solution Architecture

After reviewing the state of the art for real-time global illumination, Crassin's voxel cone tracing technique [7], seemed like a promising basis for this work. Crassin's work is based on a volumetric scene representation, in which data about any part of the scene is easily accessible, and where a single voxel can map to multiple fragments of the output image. This lead to questioning whether a voxel structure could be used to decouple the resolution at which rays are traced from the output resolution of the application. Based on this hypothesis, a technique that combines hardware-accelerated ray tracing, voxelization, and volumetric data was architected. This chapter will go over the ideas, principles, and methodologies behind this technique.

3.1 Technique Overview

As explained previously one of the main goals is decoupling the ray tracing resolution from the final video output resolution, however, it is equally a goal to reduce the overall required ray-tracing resolution to achieve a similar image quality. To do this, this work takes great inspiration from voxel global illumination techniques. The developed technique is divided into 3 stages:

- The Voxelization Stage, in which we convert the scene into a volumetric representation that encodes any data relevant to the radiance injection stage.
- The Radiance Injection Stage, in which radiance is injected into the volumetric scene representation by relying on hardware-accelerated ray tracing.
- The Final Rendering Stage, in which we take the data from the previous stage and render the scene with a normal representation and utilizing the volumetric data to shade the scene.

The pipeline generated by these three stages is illustrated in figure 3.1.

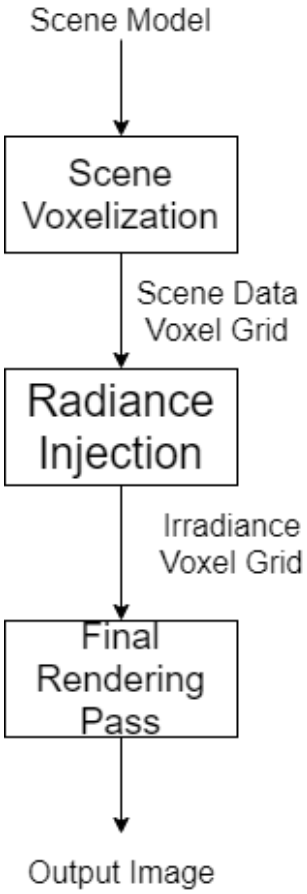

Figure 3.1: Voxel global illumination pipeline

The upcoming sections go over each of these stages, the ideas behind them, and how they work as well as any relevant caveats about them.

3.2 Voxelization Stage

Scene voxelization is the first stage in the designed solution, to that end its important to understand what voxels and voxelization are, how voxelization is done, the various data structures associated with it, and how different ways of accessing voxel data will impact the resulting image quality.

As such this section goes over the various ways in which these aspects are relevant to the proposed solution.

3.2.1 Voxel Definition

Voxel stands for volume element. In a sense they are like 3D texels, however, their usage tends to be generic and not just for texture data. Voxels, therefore, store a representation of the spatial data from a given volume.

3.2.2 Voxelization Definition

Voxelization is the process of converting a polygonal mesh or scene into a regular eulerian grid of voxels, containing relevant data for their intended usage. The simplest form of voxelization is called binary voxelization[50], where the only stored data in a voxel is whether an object is present or not in a given volume. Voxelization can also be categorized around the parts of an object that are encoded into the voxel grid, where the main categories are surface voxelization and solid voxelization[50]. Solid voxelization can be viewed as a superset of surface voxelization, and in it all parts of an object are encoded into the voxel grid, meaning the surface and the interior of an object. In contrast, surface voxelization only encodes the surfaces of objects, this means that enclosed objects in the voxel grid are hollow.

Considering that material interactions for global illumination mostly happen at the surface of objects, surface voxelization will be the only type of voxelization used in this work. However, while this work does not use solid voxelization, it could be useful to model translucent materials, the reason for this is that having easily accessible data about the interior of objects greatly helps to compute inherently volumetric data such as refractions and subsurface scattering.

3.2.3 The Advantages Of Voxel Representations

Voxels offer several benefits, in the case of this work, the most important aspects are the ability to easily obtain data regarding any coarse location in space and the ability to decouple lighting resolution from the video output resolution of the application. The latter being the main reason behind choosing to combine voxelization techniques with the hardware-accelerated ray-tracing capabilities of Nvidia's Turing GPU architecture.

In this work, it has been hypothesized that by having a single voxel map to multiple pixels in the final image, it is possible to reduce the number of rays that need to be cast to light the same area in the final image. Therefore, the lower the resolution of the voxel grid, the lower the ray tracing resolution needs to be. This being said, there will be a tradeoff in the resulting image quality, due to the final value of each pixel being tied to the voxel that it maps to. Theoretically, if there was to be a 1:1 mapping in size between

voxels and pixels, the resulting image quality would be the same; assuming any required adaptations in the rest of the rendering pipeline are made. However, since the goal of this project is to optimize render times, such a situation makes no practical sense as it would lead to increased render times, higher memory usage, among other issues.

If this technique for reducing the required ray-tracing resolution with non-stochastic ray tracing, there is no need for the usage of denoising algorithms as these serve to fill in discontinuities in the data resulting from the ray tracing process. Denoising is required to fill in the gaps when path tracing is used if the number of rays cast per frame is not enough to cover all the viewable section of the scene, and this denoising has a cost, as such not needing it should reduce render times. However, if stochastic tracing of secondary rays is desired since only a single ray is required to light multiple pixels, the effect of temporal artifacts would be reduced as the likelihood of shading discontinuities is lessened. Unlit pixel discontinuities would likely be largely absent, but lighting discontinuities could still be present and lead to temporal artifacting. However, if present the resulting temporal artifacting could be unperceivable if coupled with a temporal filtering strategy such as temporal anti-aliasing. This being said, take this what a grain of salt, since stochastic ray-tracing was not explored in the implementation this particular aspect of this chapter is merely speculation based on theory and as such should be assumed to be wrong until proven otherwise.

3.2.4 Rasterizer-Based Voxelization

There are several ways of voxelizing a scene, however, for real-time rendering, rasterizer-based voxelization has become the most popular approach. The reason for this is its simplicity and its ability to leverage the highly specialized and efficient triangle rasterization hardware present in GPUs.

The following sections discuss the basics and specific methods of performing GPU rasterizer-based voxelization, which is based on the approach presented by Michael Schwarz[50] and by an article from Nvidia[54].

3.2.4.1 Core Idea of GPU rasterizer-based Voxelization

The main idea of rasterizer-based voxelization is to map each fragment generated by the rasterizer to a specific cell in the voxel grid and writing the necessary data from that fragment into the voxel grid cell. This is done by utilizing an orthographic projection matrix that maps the input coordinates to $[0,W]$ for the width, $[0,H]$ for the height and $[O,D]$ for the depth, where W is the width of the voxel grid, H is the height of the voxel grid and D is the depth of the voxel grid. Henceforth, this projection matrix will be

referred to as the voxel-space projection matrix. After having applied this projection matrix, the x,y, and z components of the resulting position vector are divided by its w-component to calculate the homogeneous position coordinates which map to the voxel cell coordinates.

An aspect to consider is that fragments need to be generated for this procedure to work, which means that a specific projection matrix is needed to generate the positions for the vertex positions that are used as the input for the GPU-Pipeline rasterizer stage. This is also done using an orthographic projection matrix and by utilizing a viewport whose width and height matches that of the voxel grid. This orthographic projection matrix must map the vertex position to the range $[-1,1]$ for the width and height and $[0,1]$ for the depth, because these are the device normalized coordinates required as input for the rasterizer. Henceforth, this projection will be referred to as the clip-space projection matrix.

Additionally, backface culling must be disabled so that fragments that are important to the voxelization process are not discarded when voxelizing from any axis.

3.2.4.2 3-Pass Voxelization

If the voxelization process is executed exactly as described above, the voxel grid will only encode data from a single direction, which may lead to cracks and missing data in the voxel grid. There are two ways to fix this issue, however, this section focuses on the simplest approach, named 3-pass voxelization. The simplest approach is to voxelize the scene from each of the 3 major axes, therefore requiring 3 render passes. To do this we need to multiply the clip-space projection matrix by a rotation matrix corresponding to the axis we want to voxelize from, this way the fragments from the 3 view-axis are generated and we can cover cracks in the voxel grid. The voxel-space matrix does not need to be multiplied by the rotation matrix as it already maps all values properly, the rotation is only required to make sure that fragments that are not generated in one of the passes are generated in at least one the other passes.

This approach is very simple and easy to implement, however, it does require 3 render calls per rendered object, which may become very expensive as models become more complex and the cost of each voxelization draw call grows. This being said, depending on the situation some optimizations could be used to slightly reduce the cost of voxelizing the entire scene. A possible optimization would be to separate the scene into dynamic and static objects and then dynamic objects are voxelized every frame while static objects are only updated when strictly required. However, for this particular optimization since the voxel grid of dynamic objects must be cleared every frame, there must be a voxel grid just for dynamic objects, and one just for static objects. One relatively common optimization

that is possible when the underlying data structure supports LODs is to not update all LODs every frame but to instead interleave LOD updates between frames. Lastly, if none, of these optimizations, can be done the simple act of adjusting the voxel grid resolution according to the hardware's capabilities can greatly increase the overall performance of both the voxelization stage and the radiance injection stage depending on the atomic write strategy.

For simplicity and due to time constraints this approach to voxelization was taken, however, the approach that is going to be discussed next is the optimal approach for a final version of this work.

3.2.4.3 Single-Pass Voxelization

Single-pass voxelization utilizes the geometry shader to reduce the number of draw calls required to voxelize a set of objects with the same material when compared to the previously outlined solution. The geometry shader is utilized to select the axis from which each triangle has the largest overall projection and use that axis for fragment generation.

Nevertheless, this is not enough to fully eliminate cracks due to GPUs only generating fragments if a pixel's center is covered by a triangle, which leads to small cracks at the edges of triangles. The way to solve this is to employ conservative rasterization, which generates fragments as long as there is something inside the pixel and not only if its center is covered. In the past, conservative rasterization has had to be emulated using the geometry shader to expand triangles, however, ever since the Maxwell architecture "Nvidia" GPUs have had hardware support for this feature. Considering that RTX driver support is only available for Turing and Pascal architecture GPUs, it is safe to rely to on hardware support for conservative rasterization when implementing this scheme.

Unfortunately, another issue arises with the usage of conservative rasterization, as too many fragments are generated, incorrectly filling cells in the voxel grid that should be empty. While this issue can be solved via collision checking, the easiest and most effective way to get over this issue is to use an MSAA render target during the voxelization process. Nvidia[54] shows the difference between utilizing MSAA and not utilizing it.

3.2.4.4 Anisotropic Voxelization

Anisotropic voxelization improves the quality of the final result generated by the two previously described methods by turning the isotropic (non-directional) data in the voxels into anisotropic (directional) data. This improvement is particularly significative when the data encoded in the voxels changes significantly depending on the direction from which it is viewed. A perfect example of such a situation is the normals of a surface, which change

significantly between the sides of objects, in fact without this improvement the encoded surfaces normals might be completely wrong. Such is the case for very thin models where both sides are encoded in the same voxel, such as in cloth. In these cases averaging normal vectors causes them to cancel out and produce a zero-vector, which when used for lighting calculations produces incorrect results.

While this might not be as impactful on more homogeneous materials, in general, rendering relies on anisotropic data, meaning that anisotropic voxelization will in most cases lead to a significant improvement in the final rendering quality.

Another benefit of anisotropic voxelization is a significant reduction of light leaks. When isotropic voxels are used a surface has the same radiance in all 6 directions it can be viewed from, this may lead to surfaces that are facing away from the light source being lit. This problem is called light leaking and while it can happen in other situations, this is the most significant source of this issue, and, anisotropic voxelization greatly reduces it. Anisotropic voxelization solves this due to giving each face of a cube voxel its own independent value.

The way anisotropic voxelization is implemented is by storing data for each face of a voxel cube, resulting in 6 voxel grids, one from each of the 6 directions a cube can be viewed from. To store data into the correct voxel face the value to be averaged is multiplied by the dot product between the surface normal and the axis of the voxel face. Say we want to try to store a color into the negative x voxel face, the way that would be done is: $color * dot(float3(-1,0,0), normal)$. Due to the nature of the axis vector, the dot product can be simplified to $max(axisnormal, 0.0)$, for example for the negative x-axis we would do $max(-normal.x, 0.0)$. Effectively, memory usage is being increased sixfold as a tradeoff for image quality. Furthermore, due to the need for atomic averaging, the performance of the voxelization stage can also be negatively impacted.

3.2.5 Voxel Data Structures

Voxel grids must be kept in a data structure, naturally different structures have different tradeoffs, that must be kept in mind and balanced for the use case of the application.

One of the biggest considerations when storing volumetric data will be the sparse memory problem. The sparse memory problem refers to the fact that when voxelizing a scene a significant portion of that scene will be empty, however, space is still allocated for storing potential voxel data. This means that a large amount of memory is wasted, and considering that the resolution of a voxel grid has a significant impact on the resulting image quality, wasting memory can severely limit the achievable image quality on memory limited platforms.

This subsection will explore the different options for storing voxel data and their different tradeoffs.

3.2.5.1 Volumes

Volumes are the simplest way to store voxel data. They are 3D textures with the same width, height, and depth as the voxel grid. During voxelization one needs to calculate the voxel cell and write into it, no extra steps are required. Since its a texture, it is also capable of fully leveraging hardware acceleration support for 3D textures, both in terms of caching and filtering. The main issue with this structure is that it does nothing to address the issue of sparse memory. Therefore, while it is very efficient in terms of runtime performance, its memory efficiency is quite low. Texture compression could be used to reduce the impact of sparse memory, however, that would add a runtime performance cost even in texture formats that support hardware-accelerated compression and decompression (BC1-7). This performance hit would affect all stages of the pipeline and not just the voxelization stage as well. Keep in mind that this is merely speculative as texture compression is beyond the scope of this work.

This approach was taken in the implementation of this work due to its runtime performance and its simplicity, despite not optimizing memory usage.

3.2.5.2 Octrees

Octrees are a form of tree data structure that aims to fully eliminate the sparse memory problem. In this structure, each node in the tree represents an octant of the volume of its parent, and the leaf nodes contain indivisible objects, in this case, unitary-sized voxels. When there is nothing inside a volume, there is no need to store it, therefore there is no need to dedicate memory to it.

While this structure eliminates sparse memory, it gives up performance to do so. Every lookup and insertion requires a tree traversal which is nowhere near as efficient as just accessing an index in a 3D texture. Another major issue is that this structure maps very poorly to the GPU's memory model, and cannot benefit from the GPU's texture caching and filtering hardware.

Crassin[7] presents a GPU implementation of an octree which he presented in his paper on voxel cone tracing global illumination.

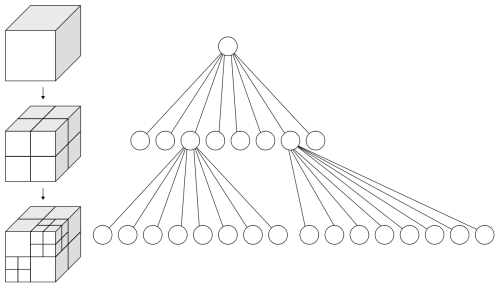

Figure 3.2: Illustration of an octree

3.2.5.3 Clipmaps

Clipmaps are a form of volume texture with embedded mipmapping. In there is something called the clipmap stack which is where the volume has greater precision and is centered on the camera. As we get further away from the camera the amount of data contained averaged into each voxel cell increases, thus reducing its precision but also increasing the volume of the scene that it represents.

This technique greatly reduces the sparse memory problem, despite not removing it fully, while giving up very little performance and hardware capabilities in the process. This structure maps very well to the GPU and is capable of utilizing its various hardware specializations. For these reasons this type of structure and similar approaches such as cascade voxel volumes have become the preferred way to store voxel data. As examples, the newest versions of Nvidia's VXGI framework uses a clipmap voxel structure[48] and Children of Tommorow[42] uses a cascade voxel grid.

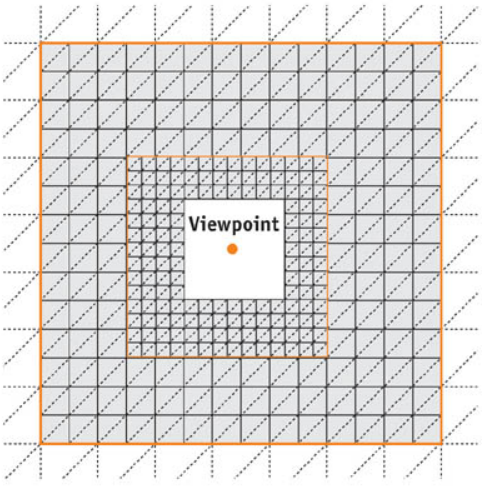

Figure 3.3: Illustration of a clipmap with its various levels of detail

For simplicity and due to time constraints this work will utilize simple volumes, however, 3D clipmaps would be the preferred way to store the voxel data this work utilizes due to their balanced tradeoff between runtime performance and memory usage.

3.2.5.4 Storing Voxel Data

An important aspect to consider during voxelization is that multiple fragments can be mapped to the same voxel cell, this means that if we merely write the fragment value to the voxel, we will create a race condition that not only will give us a less accurate voxel value, but that will create flickering during rendering since there is no guaranteed order to the fragment writes into the voxel grid.

To solve this we must rely on atomic operations to create an order-independent value calculation for the voxel. In this work, this is done via an atomic moving average. In this work since operations are done on data that is stored in vectors, usually color data, RGB channels are used to store the individual color components and the alpha channel is used to store the average's sample count. However, if we were storing data is not naturally stored over multiple channels such a single integer value, another scheme for storing the sample count must be used, an option is using only 24 bits of a 32-bit value to store the actual voxel value and use the remaining 8 bits to store the number of samples in the moving average.

This being said an atomic moving average might not be the best option in all scenarios, Doghramachi[10] presents an alternative based on atomic max operations that he uses to avoid averaging and can be used to combat performance and hardware limitations. In this he merely stores the maximum value in the voxel with a single atomic max operation, this avoids the needs for a spinlock which can greatly increase performance when there is a significant amount of threads trying to store data in a voxel. Theoretically, this is a less accurate way of storing data in a voxel, however, it can be a significant performance improvement for what is being given away, especially if we consider than as long as the voxel grid's resolution is not too low, the reduction in accuracy is very slight.

3.2.6 Voxel Filtering

Radiance is encoded in a discrete grid of values, that means that without any kind of processing, the final shading will have the grid being visible, some areas more than other, but there is no way to escape this issue, even normal diffuse textures have to deal with this issues. So to solve it, we take a page from the book of how we improve texture quality, rely on interpolated values. By utilizing trilinear interpolation when sampling the values from the grid we can make the grid invisible when rendering, this is because we have interpolated the values between voxel cells, creating a smooth gradient. For sampling to be correct we use the floating-point coordinates that we get back from applying voxel-space projection matrix, instead of rounding them and converting to integers. Since GPUs have hardware accelerated trilinear filtering, this simple image quality improvement technique

has a negligible cost, however, even if it was not negligible the resulting image quality would be too unsatisfactory to use without.

3.2.7 Relevant Data To Store

Depending on the diffuse and specular BRDF's in use the data that must be kept in the voxel grid will vary, in here we will outline the data that would be stored and its format for two different popular material models.

An important aspect to consider is that we only actually need to store material data that might be required in some way during the light injection stage. The irradiance map used to store the radiance value in each voxel is assumed to be required in all of the material models and as such will not be discussed in the sections below. The same idea is expected for emissive materials, as data on emissive materials would be store in the same way for both the Blin-Phong model and a microfacet material model or PBR material model as it is more commonly known.

3.2.7.1 Blin-Phong Material Model Data

For this model, we need to store in the voxel grid every aspect about the material as it will be needed either for primary rays or secondary rays. Since we are required to calculate the dot product between the light direction and the surface, we will need to store the surface normal in the voxel grid. We also need to store the actual specular and diffuse coefficients due to the need to multiply them by the correct dot product, an option to get around this would be to store the radiance multiplied by each of the dot products in a different voxel grid, however, this would only eliminate the need to store the specular coefficient and would result in no difference in terms of memory usage. The diffuse coefficient needs to be stored so that it can be used for the radiance of the secondary rays. The specular exponent must also be stored so that the dot product between the light reflection and the view direction can be exponentiated during light injection, and since this is the only stage where this can be calculated correctly there is no way to get around needing to store this data.

Table 3.2 summarizes all data that must be stored, the format it is stored in, and any relevant notes about it.

A possible approach to not have to compute as many elements in this stage of the pipeline is to have a map where you store in one channel the dot product between the light direction and the surface normal and in another the dot product between the light reflection direction and the view direction. This would eliminate the need to use a voxel grid storing the specular coefficient and the specular exponent, however, you would still

Data	Format	Notes
Surface Normal	RGBA8 Unorm	Alpha Channel Contains Average's Sample Count
Diffuse Coefficient	RGBA8 Unorm	Alpha Channel Contains Average's Sample Count
Specular Coefficient	RGBA8 Unorm	Alpha Channel Contains Average's Sample Count
Specular Exponent	R32 Float	24 Bits Contain Actual Exponent, Remaining 8 Contain Average's Sample Count

Table 3.2: Required voxel grids for Blin-Phong material model data

need a map for the dot product data and you would still need to compute light falloff data in this shader. The diffuse coefficient data would still be needed to be able to reflect the material color through the secondary rays, and the map containing the surface normals would naturally also still be needed.

Table 3.3 illustrates the voxel grid used in this scenario.

3.2.7.2 PBR Microfacet Material Model Data

Microfacet-based material models can change significantly between each other in how they can calculate the actual color value for each fragment, however, some data is common between all of them and this specific section is based on the material model used in the Frostbite engine[30].

Frostbite makes use of the Disney diffuse term with energy renormalization for the diffuse component and uses the Smith correlated visibility function and GGX normal distribution function for the specular component. As such materials in this model have an albedo or base color, a smoothness parameter, a metal mask parameter, and a reflectance parameter. Since smoothness, reflectance, and the metal mask are single values we can pack them into a single voxel cell using 3 channels of a float4 and the final channel for the average's sample count. The base color is given its voxel cell. And just like before the surface normal must also be stored. However just like before we can store just the dot products so that we do not have to compute as much data in the light injection stage. While this reduces computations in the light injection stage it does not affect overall memory usage. Regarding the dot products in this model, we are required to store 3 different dot products, not 2 like in the Blin-Phong model. We need to store the dot product of the normal with the view direction, the normal with the light direction, and the light with the half vector between the view direction the light direction.

Table 3.4 illustrates the voxel grids required to store the material data under this scheme.

Data	Format	Notes
Surface Normal	RGBA8 Unorm	Alpha Channel Contains Average's Sample Count
Diffuse Coefficient	RGBA8 Unorm	Alpha Channel Contains Average's Sample Count
Dot Product Storage	R11G11B10 Float	The Red channel contains NdotL, Green Channel contains LdotV and the Blue channels stores the average's sample count

Table 3.3: Required voxel grids for Blin-Phong material model data under the proposed scheme to reduce memory usage and computations during the light injection stage.

Data	Format	Notes
Surface Normal	RGBA8 Unorm	Alpha Channel Contains Average's Sample Count
Albedo	RGBA8 Unorm	Alpha Channel Contains Average's Sample Count
Dot Product Storage	RGBA8 Unorm	The Red channel contains NdotL, Green Channel contains LdotV, Blue channel contains VdotH and the Alpha channel contains the Average's sample count

Table 3.4: Required voxel grids for Frostbite material model data under the proposed scheme

3.3 Ray Traced Radiance Injection

The light injection step is responsible for injecting light into the voxel grid and propagating it to its correct positions. This step handles both direct lighting injection and indirect lighting injection.

This step makes use of hardware-accelerated ray tracing to propagate light rays through the scene and inject them into the correct cells in the voxel grid. In general, two approaches can be taken to this, ray tracing from the light source and ray tracing from the camera. This section will explore how these two approaches could be implemented and any relevant caveats about them.

The subject of acceleration structures will also be touched upon as these are very important for the actual ray tracing process. This being said, since much of the work related to these is done by the GPU driver this work merely focuses itself on the idea of turning the voxel grid into an acceleration structure and using that versus the traditional geometry-based acceleration structures. Do keep in mind that a deep dive investigation into all aspects related to acceleration structures is beyond the scope of this work. All discussions presented on this topic serve to spotlight the importance of this topic, both from an architectural and an implementation point of view.

3.3.1 Ray-Tracing From The Light Source

In a sense this is the easiest way to do ray tracing, it was also historically the first to arise, however, it is not optimal. There are two key issues with this approach:

The first issue is that many of the rays traced will in no way contribute to the visible lighting, which means performance is being needlessly wasted.

The second issue is that different light types will require more or fewer rays to be cast from it. A spotlight only needs to cast rays in a given direction, however, a point light needs to cast rays all around it, assuming we want on average the same number of rays to hit each voxel, the point light is significantly more expensive to use than the spotlight. Additionally, in DXR it is significantly harder to properly implement a point light without wasting rays when compared to a spotlight. We will touch in this in chapter 4, but DXR's ray casting API seems to favor casting rays either from a plane's or the camera's perspective.

One issue that both models will face under this work is that which this work refers to as voxel ray overloading, what this problem refers to is that to write to a voxel we rely on a moving average that uses a spinlock. When many rays hit the same voxel, the amount of time per-ray that spent on the spinlock for the moving average increases substantially, creating dramatic changes in render-time for this stage of the rendering pipeline. In the case of ray tracing from the light source, this happens when a light is very close to geometry, leading to more rays hitting the same voxel. Increasing the resolution of the voxel grid helps alleviate this issue, however, it can also cause cracks in the lighting voxel grid depending on the ray-tracing resolution and will also increase the cost of the voxelization stage.

The one advantage of this model is that its significantly easier to use in a prototype when compared to ray tracing from the camera's perspective, especially if spotlights are

more than enough for testing. This is the reason that this model is used in the implementation of this work, however, ray tracing from the camera is surmised to be the optimal approach.

In this model when a ray hits a surface, we can use the distance traveled by the ray to calculate the attenuation of the light intensity. We convert the position at which we hit the geometry to compute the corresponding voxel grid cell, by using the same voxel-space projection matrix that was used during the voxelization stage. We can then compute the dot product between the surface normal and the light direction, the dot product between the light reflection direction and the camera view direction and the dot product between the halfway vector and view direction vector, this is the only stage of the pipeline that can compute this particular set of data. The rest of the computations regarding the BRDF can either be computed here which gives us per voxel data or in the final stage which will give us per-pixel data.

This covers the data that is computed when a ray hits a surface, for indirect rays we merely need to trace a new ray from the position at which we hit the surface and update the ray payload with any required changes such as light color and intensity. The direction of the new ray is merely the previous direction of the light ray reflected by the surface normal, this is if we are merely doing ray tracing. If we are doing path tracing; which is not recommended as this work does not explore temporal coherency under path tracing due to it being stochastic; we take a random direction either being importance sampled or not and trace the new ray in that direction. When a secondary ray hits a surface we use the same procedure to calculate the impact of this ray on the lighting grid as was used for direct rays.

In general, the most expensive aspect of this whole process will be the atomic averaging of voxel data. There are no real-ways of eliminating the impact of atomic averaging, however, it can be reduced by balancing the ray tracing resolution and the voxel grid resolution. Another option would be to not use atomic averaging, instead substituting it by atomic maxima, this can either be done for all stages of just in the radiance injection stage.

3.3.2 Ray-Tracing From The Camera

Ray tracing from the camera, appeared as a response to the waste of rays when ray-tracing from the light source. In this, we trace rays from the camera into the scene, and when a ray hits a surface we then trace a ray in the direction of each light to understand if that particular surface point is being lit by the light, these second set of rays are called shadow rays. If secondary rays are required we then send a reflected ray from the original hit

point into the scene and if we hit a point we once again cast shadow rays and count the contributions of those lights to the overall lighting of the original point.

For calculating the dot products we can use the direction of the shadow ray. The rest of the process is the same as ray tracing from the light source.

The main advantage we get from this scheme is that all rays cast contribute to the viewable lighting of the scene, we don't spend time calculating lighting for parts of the scene that are not visible. The other important improvement is that all lights have a very similar cost since we are not emitting rays from the light itself, we are merely checking if the light is visible from the surface point.

Regarding the voxel ray overloading problem, this model still suffers from it, but in a different scenario. When the camera is very close to lit geometry, there will be a significant amount of rays hitting the same voxel and calculating different values for it, making the spinlock run for longer. This is the main issue that both models face.

While camera ray-tracing is not implemented in this work, it is expected to be the best option. The main reason for this is feature support as a significant amount of approximations developed rely on the usage of shadow rays or take advantage of rays being cast from the camera. Performance is also a significant reason to prefer this option, however, quantifying the exact impact of this aspect is complicated as it will depend on the geometry, camera position, and point of view.

3.3.3 Acceleration Structures

Acceleration structures are fundamental to accelerating the ray tracing process in traditional software-based ray tracing, but in hardware-accelerated ray tracing not only are they fundamental they mandatory to use. These allow us to predict when a ray will not hit any relevant geometry and therefore accelerate the detection of ray misses.

For hardware-accelerated ray tracing, the GPU driver is responsible for creating an acceleration structure from input data, and in this case, they will use bounding volume hierarchies, which are commonly abbreviated to BVH.

3.3.3.1 Bounding Volume Hierarchies

BVH's are traditionally implemented with a tree where each node represents a volume with some amount of objects inside it, as you go deeper into the tree, the volume represented by each node is smaller and smaller until you reach the leaves which represent the volume occupied by a single object. When tracing rays you can intersect rays with the top-level nodes, if there is no intersection you know there is no point in going down that path of the tree and if you don't intersect with the root node at all, you can immediately

discard a ray as having missed the geometry. This structure therefore greatly speeds up the ray tracing process, however, there is a cost to building it and updating it. In the case of hardware-accelerated ray tracing, we know a BVH type acceleration structure is used, however, we don't know how it is implemented or how it works as that is handled by the GPU-driver. This being said, the fact that there is a cost to building and updating this structure is still true and is something that application's utilizing hardware-accelerated ray tracing must manage.

3.3.3.2 Voxel BVH vs Geometric BVH

An interesting option to consider is that of building the ray tracing acceleration structure based on a voxel representation of the scene, compared to the regular geometric representation of the scene. This work uses a traditional representation of the scene and not a voxel-based one, however, a voxel representation was attempted and successfully used, however it came with disadvantages that did not make sense to try to overcome in the timeframe available for this work.

The assumption behind a voxel-based acceleration structure was that it would always be simple to intersect rays and that due to only needing a single object primitive and all other things are instances of the same object. In practice building the acceleration structure took significantly longer for the driver; depending on the size of the voxel grid. There is also a significant hit from having to copy the voxel grid data from the GPU to the CPU and then having to build the acceleration structure, which would incur another data transfer hit from having to transfer data from the CPU to the GPU. Lastly, it incurs an accuracy penalty when intersecting rays with the geometry, specifically with rays that have a negative direction. Due to the ray intersection being at the boundary between voxels when converting to homogeneous voxel coordinates the calculated cell is one voxel behind in one direction or more of the actual voxel that should have been picked, which would require a scheme to fix this, for no gain whatsoever.

On another note, early tests also showed that ray intersection took longer for the voxel acceleration structure when compared to the geometrical acceleration structure, however, these results were not analyzed extensively, but did contribute to the overall decision to move away fro ma voxel-based acceleration structure.

So due to all these aspects and due to time constraints a geometric acceleration structure is used, however, it is to be noted that this scene is fully static, so we never actually need to update the acceleration structure. Investigating the cost of updating a voxel acceleration structure when compared to a geometric acceleration structure, is an interesting

road to go down, especially for more dynamic scenes and objects whose shape morphs during runtime.

3.4 Final Pass

The final rendering pass is by far the simplest, as it only needs to bring together the data computed in the previous steps. The exact data that needs to be computed will vary based on the BRDF what we chose to compute during the lighting injection stage, however in general the amount of computations for a basic final pass shader is very small.

3.4.1 Calculating The Final Color

The first thing that needs to be done during this stage is to calculate the voxel cell that needs to be accessed for lighting information during this stage. This is done by taking the position of the vertex and multiplying it by the voxel-space conversion matrix described in section 3.2. After that, we have the homogeneous coordinates of the voxel grid cell that corresponds to the fragment and we can use that any data encoded in the voxel grid that is required to compute the final fragment color.

If during the light injection stage almost all of the BRDF was calculated we only need to read the irradiance and multiply it by the texture color for that fragment and add to it any ambient light we desire and that gives us our final fragment color. However, if we only encoded the dot products related to our material model, we will have to read those and compute the final fragment color using the BRDF's. Once again these will depend on the material model being used, but in this work, the Blin-Phong material model is used.

3.4.2 Gamma Correction And Low Dynamic Range

One final aspect that needs to be considered is that of gamma correction and conversion to a low dynamic range if such is necessary. In the case of this work, radiance is encoded in an 8-bit per channel voxel grid, meaning that it is already in a low dynamic range, however, for higher accuracy, it could be encoded in a higher precision format, this means that it would have to be converted into a lower precision format for monitors that only support 8-bits per channel, this is done via tone mapping, unfortunately, this is something that is not explored in this work as it does not impact the way this technique would behave.

Another important aspect to consider is that of gamma correction, while this also does not affect how this technique works, it does affect its perceived quality. Gamma correction serves to account for the lower precision of 8-bit per channel render targets and

is based on the fact that the human eye has a hard time perceiving difference at higher light intensities. What is done here is a non-linear remapping of colors to better spread the values between 0 and 1 such that they allow the human eye to better perceive gradual changes. Image 3.4 illustrates the difference between a linear color space grayscale and a gamma correct color space grayscale.

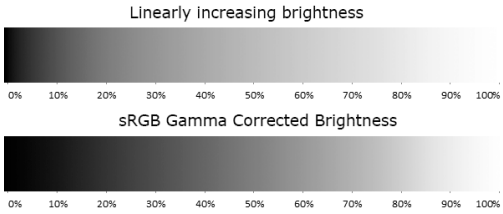

Figure 3.4: Linear Grayscale compared to a Gamma-Correct Grayscale

Eric Lengyel[33] presents a great overview of this topic and how it works. In the case of this work, we merely need to make sure that textures that are not in the sRGB format when created are converted from the gamma-correct non-linear color space into a linear color space before doing lighting calculations with them, and that if the render target is not using the sRGB format we gamma correct the final fragment color value before outputting it. In this case, we use the gamma correction functions presented by Eric Lengyel[33]. Those are:

$$f_{sRGB}(c) = \begin{cases} 12.92c & c \leq 0.0031308 \\ 1.055c^{1/2.4} - 0.055 & c > 0.0031308 \end{cases}$$

$$f_{linear}(c) = \begin{cases} \frac{c}{12.92} & c \leq 0.04045 \\ (\frac{c+0.055}{1.055})^{2.4} & c > 0.0031308 \end{cases}$$

where f_{sRGB} takes a component of a color from a linear color space to the gamma correct sRGB color space. f_{linear} on the other hand takes a component of a color from the gamma correct sRGB color space to the linear color space.

Chapter 4

Solution Implementation

Chapter 3 present the theory and architecture behind this work, this chapter will focus on an implementation of that architecture. Unfortunately, due to the time constraints not all aspects of the architecture were possible of being implemented, these have already been specified in chapter 3.

4.1 Technological Choices

For this work API level support for hardware-accelerated ray tracing is a necessity, therefore the only natural options are Direct3D 12 and Vulkan. Due to Vulkan Ray Tracing having been released very recently at the start of this project, Direct3D 12 became the natural choice, especially considering the author's previous experience with this API. Unreal Engine 4 was explored as a possible testbed for this project, however, it quickly became apparent that it would be a bottleneck for development rather than simplifying it, therefore a fully custom rendering engine based on DirectX12 has been developed for this work instead. TinyOBJLoader was used to assist with model loading and Dear ImGui for the debug interface utilized for development.

4.2 Voxelization Stage

Scene voxelization is necessary to be able to access the data of specific voxel during the light injection stage. This is done by voxelizing any relevant scene data and storing it in data structures that will then be passed on to the light injection stage. Originally, the Nvidia VXGI SDK was explored to handle this aspect of the rendering pipeline, however, it quickly became apparent that it would not be an option as VXGI did not expose the relevant resource data required to use the voxelization data inside the light injection stage,

Data	Resource Format	UAV Format	SRV Format
Diffuse	R8G8B8A8_TYPELESS	R32_UINT	R8G8B8A8_UNORM
Specular	R8G8B8A8_TYPELESS	R32_UINT	R8G8B8A8_UNORM
Specular Exponent	R32_FLOAT	R32_FLOAT	R32_FLOAT
Normal	R8G8B8A8_TYPELESS	R32_UINT	R8G8B8A8_UNORM

Table 4.1: Formats of the resource, UAV, and SRV of the various voxel volumes used in this work. Note that all of these formats are prefixed by DXGI_FORMAT_ which has been omitted.

as well as other hard to manage aspects of that library, as such a custom voxelization phase was developed and VXGI was removed from the project.

This section explores how scene voxelization was implemented for this project.

4.2.1 Voxel Grid Creation

Due to time constraints and for simplicity voxel grid in encoded in a simple 3D Textures(Volumes) instead of in 3D clipmaps. To create a volume for the voxel grid we must create a resource in the GPU that will hold its data, in D3D12 that is done via an ID3D12Resource which is a COM interface to a GPU resource.

An important aspect to consider when creating these resources is that they must support atomic reads and writes, however, atomic operations are only supported on integer data formats. This creates an issue because hardware-accelerated texture sampling is not available for integer data formats in DirectX, it is only available for floating-point data formats. This would indicate that to have one feature we would have to give up the other, however, that is not the case, we can create the resource without specifying its data format with the TYPELESS identifier. We can then create an SRV with a floating-point data format allowing us to sample from the texture, and when we create the UAV with an integer data format, giving us the ability to execute atomic writes into the texture. Table 4.1 specifies the D3D12 data format for the resource, SRV, and UAV which are used in this work for each of the voxel grids.

Regarding the flags used during resource creation, we must provide it the flag that tells it we want to be able to write to it via a UAV. Additionally, we must give it the flag that places the resource in a default heap which does not allow reading or writing to it from the CPU. Placing a resource in a default heap over one that is CPU visible is a significant speed improvement and should be leveraged as much as possible. The rest of the code is of little to no relevance to this work as it is standard code used in the creation of D3D12 resources, and has no notable changes for this work.

Code Listing 7.1 illustrates how to create the voxel volume resource in D3D12.

4.2.2 Voxel Volume SRV Creation

To be able to sample the voxel volume in the shaders, a shader resource view into the voxel volume must be created. In D3D12, this is done by filling out a structure that describes the resource view and requesting the device to create a view with the supplied configuration. Views to resources (also called descriptors) must be kept in a special heap that maps the various views to a resource to the actual resource. These heaps have to be explicitly managed by the application, therefore we have had to build a system that is responsible for allocating indexes into these heaps. In this case, you will see the code listing request the shader resource heap manager to allocate a slot for it in the heap, these heaps managers are not the subject of evaluation of this work, but for ease of understanding, these are very simple, they merely have heaps with a predefined size and a list of empty slots into it that it uses for the actual allocation process.

Keep in mind that the shader resource view must specify the floating-point data format for the voxel volume to be able to sample from it.

Code Listing 7.2 demonstrates the creation of a shader resource view into the voxel volume.

4.2.3 Voxel Volume UAV Creation

To be able to write into a texture we need to create an unordered access view into it, similar to the case of shader resource views. This is done by filling out a D3D12 structure and requesting the device to create a view based on that and stored it into a descriptor heap. The only meaningful change here is that not only do we store the view in the normally managed descriptor heap, we also store it in a descriptor heap that is not visible in shaders and that is not managed. This allows us to clear the voxel volume through this unmanaged descriptor heap.

Keep in mind that if you need to write atomically into this texture, this UAV must use an integer data format.

Listing 7.3 demonstrates how to create an unordered access view into the voxel volume.

4.2.4 Voxelization Matrices

The clip-space projection matrix and voxel-space conversion matrix are defined based on an orthographic projection as specified in chapter 3, however, there is one additional aspect that needs to be accounted for. As a scale of 1, a model or scene can bee too large to fit in the view volume or the voxel grid, or too small, leading to a loss in precision. This

means that the models must be scaled to appropriately fit the desired coordinate range. The easiest way to account for this is to compute the axis-aligned bounding box of the model or scene and calculate the extent of it. Afterward, based on this data, and on the coordinate range that the model must be scaled too the scale can be normalized to fit the desired coordinate range despite the case which needs to be addressed.

4.2.5 Voxelization Draw Call

This works utilizes the significantly simpler 3-pass voxelization scheme to voxelize the scene, to that end it refers to a voxelization draw call as voxelizing the scene from a single view axis. Each voxelization draw call receives a constant buffer with its voxel-space projection matrix and clip-space projection matrix, recall that the only matrix that changes with the view-axis will be the clip-space matrix. This is the only data that will change in between the 3 draw calls required to voxelize a set of objects with the same material.

Models are drawn and voxelized in submeshes of the same material. The underlying vertex buffer is the same, however, the index buffer is changed to only draw the objects with the same material. In this context, a model can be made up of several sets of objects with the same material and there will be a set of 3 draw calls for each of those materials. With larger models and more complex scenes, this part of the technique can become very expensive and is the reason why the single draw call approach tends to be preferred.

Listing 7.4 illustrates how a set of objects with the same material is voxelized, where XAxisVoxelizationCall is the function that issues a single voxelization draw call with the constant buffer set such that the clip-space projection matrix voxelizes the scene as viewed from the x-axis. Listing 7.5 shows how a single voxelization draw call is done, in this case from the x-axis.

4.2.5.1 Removing Sign From Normals

An interesting aspect to consider for storing normals is that they are signed vectors, however, we want to store them as unsigned color values. For this reason, we have to convert them into unsigned values to guarantee there are no numerical issues. To do this we need to compress and shift the normal values by dividing them in half and moving them forward by 0.5, to shift the value from the -1 to 1 range to the 0 to 1 range. When we want to read them we need to multiply the value by 2 and subtract 1 to put them back in the original -1 to 1 range.

This can be observed in the vertex shader of listing 7.6.

4.2.5.2 Packing RGBA8 Unorm Into 32-Bit Unsigned Integer

To be able to write atomically into a texture, that texture must be in an integer format, however, out data tends to be in 8-bit per channel float format. This means that we need to pack a 4 channel float into a single 32-bit unsigned integer value. This is relatively simple and involves shifting values to their specific positions. One thing to keep in mind is that we do need to remap the original float values since our values are in the 0 to 1 range and an 8-bit integer stores values in the 0 to 255 range, to do this we merely need to multiply the floating-point value by 255. Since the values are in the 0 to 255 range when we unpack them to get back the original value we need to divide them by 255 to get back the original value in the 0 to 1 range.

Listing 4.1 demonstrates how to pack a 4 channel float into a 32-bit unsigned integer in HLSL. Listing 4.2 demonstrates how to unpack a 4 channel float from a 32-bit unsigned integer in HLSL.

```
1
2  // Packs float4 in [0,1] range into [0-255] uint
3  uint PackFloat4(float4 val)
4  {
5      uint r = round(clamp(val.r, 0.0, 1.0) * 255.0);
6      uint g = round(clamp(val.g, 0.0, 1.0) * 255.0);
7      uint b = round(clamp(val.b, 0.0, 1.0) * 255.0);
8      uint a = round(clamp(val.a, 0.0, 1.0) * 255.0);
9      return (
10             (uint(a) & 0x000000FF) << 24U |
11             (uint(b) & 0x000000FF) << 16U |
12             (uint(g) & 0x000000FF) << 8U |
13             (uint(r) & 0x000000FF));
14 }
```

Listing 4.1: Packing a 4-channel float into a 32-bit integer

```
1
2  // Unpacks values and returns float4 in [0,1] range
3  float4 UnpackFloat4(uint val)
4  {
5      uint r = (val & 0x000000FF);
6      uint g = (val & 0x0000FF00) >> 8U;
7      uint b = (val & 0x00FF0000) >> 16U;
8      uint a = (val & 0xFF000000) >> 24U;
```

```
9     return float4(r / 255.0, g / 255.0, b / 255.0, a / 255.0);
10 }
```

Listing 4.2: Unpacking a 4-channel float from a 32-bit integer

4.2.5.3 Moving Average

Since multiple fragments can map into the same voxel cell during voxelization, we need a scheme to account for this. The most accurate way to handle this is to average all the values together via a moving average. However, this is easier said than done, since there is no guarantee of the order in which fragments will try to write to the voxel. We need to execute the moving average atomically, that is the reason why we need to pack the floating-point values into integers. HLSL has a set of intrinsics related to atomic operations, however, in the case of a moving average, the only one of interest is *InterlockedCompareExchange*. The *interlocked* prefix denotes that its an atomic operation, *Compare* means that it will atomically compare with the current value in the texture and *exchange* means it will exchange the values. Therefore, what this function does is check the current value of the texture cell with the comparison value we provide and if the values are the same it will store a value we provide into it and return the value that was previously there. If the values being compared are not the same it will return the currently stored value. Relying on this function to implement a spinlock is fundamental to create an atomic moving average.

```
1
2  void AverageRGBA8Voxel(RWTexture3D<uint> voxel_map, int3 voxel_coords,
       float4 val)
3  {
4      uint packed_color = PackFloat4(float4(val.rgb, 1.0 / 255.0));
5      uint previousStoredValue = 0;
6      uint currentStoredValue;
7
8      float4 currValue;
9      float3 average;
10     uint count;
11
12     InterlockedCompareExchange(voxel_map[voxel_coords],
           previousStoredValue, packed_color, currentStoredValue);
13     while (currentStoredValue != previousStoredValue)
14     {
15         previousStoredValue = currentStoredValue;
16         currValue = UnpackFloat4(previousStoredValue);
```

```
17
18          average = currValue.rgb;
19          count = round(currValue.a * 255.0);
20
21          average = (average * count + val.rgb) / (count + 1);
22
23          packed_color = PackFloat4(float4(average, (count + 1) / 255.0));
24          InterlockedCompareExchange(voxel_map[voxel_coords],
                previousStoredValue, packed_color, currentStoredValue);
25      }
26  }
```

Listing 4.3: Atomic moving average of the voxel values

4.2.5.4 Complete Voxelization Shader

Code listing 7.6 represents the entire shader that is used during a single voxelization call, with both the vertex shader and pixel shader being in the same listing, however, functions that have already been demonstrated have been removed for the sake of brevity.

4.3 Radiance Injection Stage

As previously explained, this work relies on a hardware-accelerated ray tracing step to inject radiance into the voxel grid. This section goes over the implementation aspects of that step. Since this work uses DirectX, implementation details are guided around the DirectX Ray Tracing API specification, however, the only relevant aspects of that specification for this work are the acceleration structure and the new shader stages added to support ray tracing.

4.3.1 Acceleration Structure

As previously explained, an acceleration structure is fundamental to achieving real-time ray tracing. In DirectX ray tracing, the GPU driver is responsible for generating an acceleration structure bounding volume hierarchy from the supplied scene description.

4.3.1.1 BVHs in DirectX Ray Tracing

In DirectX ray tracing, a bounding volume hierarchy is organized into the form of a depth-two tree. At the root of the tree is the BVH acceleration structure. At depth one lies the

top-level acceleration structure instances, or TLAS instances, and each one of these then references a bottom level acceleration structure or BLAS. Each BLAS must be built by the GPU independently, however, TLAS instances are not built individually but rather all together to create the final acceleration structure. Both types of acceleration structures are built by filling out a D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_INPUTS structure instance with the information corresponding to the level of acceleration structure being built, as well as D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_DESC structure instance. Building any of the types of acceleration structure will require the usage of both a scratch buffer and a final buffer to hold the structure, therefore, these must be created in the default heap. The amount of memory that each of these must hold can be queried by filling out a D3D12_RAYTRACING_ACCELERATION_STRUCTURE_PREBUILD_INFO structure and calling the GetRaytracingAccelerationStructurePrebuildInfo() function of the device.

Building the actual acceleration structure is done by calling the BuildRaytracingAccelerationStructure() function in the command list, however, these structures are built as unordered access views and the program must wait for the built to be complete before using them. To wait for the build to finish, a UAV resource barrier is used, which just waits for all accesses to the acceleration structure to end before allowing the program to continue.

Bottom Level Acceleration Structure Bottom level acceleration structures represent individual pieces of geometry, be it a single cube or an entire model. To specify one the D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_INPUTS structure is supplied with the D3D12_RAYTRACING_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL type flag and by supplying it with instances of the D3D12_RAYTRACING_GEOMETRY_DESC structure. The D3D12_RAYTRACING_GEOMETRY_DESC structure describes the object contained inside the BLAS, it takes a data about the vertex buffer, data about the index buffer, a transform, and a specification of whether or not the BLAS is opaque. In the shortest possible sinse it is the main description of the BLAS. D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_INPUTS also receives flags about how to build the acceleration structure, these impact several aspects of the structure, however, in our case the only flag that was ever used was D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_PREFER_FAST_TRACE which is meant to improve the performance of the ray-tracing process.

Top-Level Acceleration Structure Top level acceleration structures are a form of instancing, as used traditionally in rasterization, to efficiently duplicate the same geometry

with different transforms. In this case, TLAS instances are pointed to a BLAS which they refer to and a transform that must be applied, allowing the program to easily add versions of the same object without needing to have several BLAS representing the same object but with different transforms. A TLAS is specified by passing the D3D12_RAYTRACING_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL type flag to the D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_INPUTS structure and the input for that structure will be the definition of each of the TLAS instances contained in the TLAS, the D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAG_PREFER_FAST_TRACE build flag is also used here. TLAS instances are described through the D3D12_RAYTRACING_INSTANCE_DESC structure, which takes the GPU address of the BLAS it points to and the transform of the instance. The rest of the process is fundamentally the same as what is done to build the BLAS.

Final Considerations Before moving on from this topic, it is important to go over some aspects to kept in mind about what was discussed and how it was used in this project. First of all, how the acceleration structure build was discussed is very much a surface level explanation there are many other aspects to consider and this overview does not substitute the official DirectX documentation and specification in any way. Another aspect to consider is that there has been little to no time invested in understanding how the various flags that can be used when building the acceleration structure affect the final image quality and performance. Finally, in this work the way that BLAS and TLAS are used is very simplistic due to the inherently static nature of the test scenes in use, a single BLAS is created for the entire scene geometry and only a single TLAS instance is used to point to this BLAS. Since the scene is static, acceleration structure rebuilds, updates, and refits have in no way been explored either.

4.3.2 Radiance Injection

The radiance injection stage relies on DirectX ray tracing to propagate light rays throughout the scene and inject their radiance into the voxel grid. Thise section, will go over relevant aspects about the DirectX ray-tracing pipeline as well as any notable aspects of how this step was implemented.

4.3.2.1 The Ray Tracing Shaders

DirectX Ray Tracing(DXR) adds a new shader pipeline dedicated to supporting ray tracing, made up of 5 shaders, however, pipeline might not be the best name for it as these shaders are not ran sequentially. The shaders that run depend on whether or not a ray

hits a surface and where. Chapter 1 and 2 of Ray Tracing Gems[16] presents an excellent overview of how this pipeline works, however, this section will briefly explain it before moving onto explaining the work done in each of those shaders in this project.

The first of the 5 added shaders is the ray generation shader, or the raygen shader for short. This shader is the entry point of the ray tracing pipeline and its main job is to generate rays to be traced. Rays are traced via the TraceRay intrinsic which will block the ray generation shader until the ray tracing process for that ray is complete. Rays carry a payload, which can be used to transport data between the ray generation shader and any other ray tracing shader that runs on that ray. The amount of raygen shader dispatched corresponds to the specified ray tracing resolution when dispatching the ray-tracing pipeline. For example, if a resolution of 512x512x1 was specified, then 262144 instances of the raygen shader would be generated.

There is a shader called the intersection shader which handles the intersection data of the ray and the scene. This shader can be the subject of very deep conversations and implementation details, however, its usage is not necessary for this work and as such it has not been explored. Therefore, the default intersection shader is used.

When a dispatched ray completely misses the geometry, the miss shader is called for that ray. What is done in this shader will depend on what the application wants to do when a ray misses the scene.

When a ray hits the scene, two types of shaders can be called, the closest-hit shader and the any-hit shader. If we trace a ray through a scene it will usually intersect with several points. The any-hit shader is run for each one of these points (unless a flag is used to stop at the first hit), this is done while the GPU is looking for the point that is closest to the ray origin. When the point that is closest to the origin of the ray is found, the closest-hit shader is called for that point. This represents the first surface point which the ray hit and is where and lighting calculations must happen.

4.3.2.2 Ray Generation Shader

For this work, the raygen shader will not only trace rays but also be responsible for calculating radiance after the TraceRay call returns, however, this section will only focus on the ray generation aspect, radiance calculation and injection will be covered later.

Since in this work ray tracing is executed from the light source and not from the camera, the implementation that will be shown is for a spotlight as that is far simpler to calculate.

The direction of the ray is calculated rather straightforwardly, we merely take the index of raygen shader thread and subtract the halved ray-tracing resolution from it, this

effectively makes is to where rays will be sent in the negative x-direction. However, there is nothing special to this process, and any other way to calculate this would be just as valid. The only common factor that all direction calculations must have is that they must account for origin shader thread to chose the proper direction unless you want them all to go in the same direction. The origin of the ray will be the position of the light, this is why we want all rays to have different directions because they come from the same point in space, however, if you varied the ray origin per ray, it could be perfectly valid to have all rays have the same direction. In general, calculation of the ray origin and direction is not an important aspect of this project. Listing 4.4 demonstrates how the ray direction is calculated.

```
1 float3 direction = float3(DispatchRaysIndex().xyz) - float3(
      X_RT_RESOLUTION/2, Y_RT_RESOLUTION/2, Z_RT_RESOLUTION/2);
2 direction = normalize(direction);
```

Listing 4.4: Computing ray direction

Recall that rays carry with them a payload used to transfer data between the various shaders. Listing 4.5 illustrates the structure that defines the data contained in the ray payload for this project. color is the color or radiance of the light ray. Initially, this is the value set by the light, however, secondary rays take on the properties of the materials they have interacted with. Origin specifies the position where the ray hits a surface. Direction is the direction of the incident ray reflected on the surface. Distance is the distance the ray traveled before hitting the surface, this value is used to light attenuation. Finally, missed is a boolean telling us if the ray hit a piece of geometry or not.

```
1 struct RayPayload
2 {
3     float4 color;
4     float3 origin;
5     float3 direction;
6     float distance;
7     bool missed;
8 };
```

Listing 4.5: Ray payload structure

4.3.2.3 Miss Shader

For this work the miss shader does nothing special, it merely sets the ray payload missed boolean to true. However, in more complex scenarios, it could for example sample from an environment map, what would be done would depend on the goals of the application, however, for this work no such behavior is necessary.

4.3.2.4 Closest-Hit Shader

For this work, the closest-hit shader is responsible for computing 2 things: the position where the ray hits a surface and the direction of the reflection ray.

The position where the ray hit a surface is computed by taking the distance traveled by the ray, which is the return value of the RayTCurrent() intrinsic, multiplying that by the direction of the ray and adding that to the ray origin. This process gives us the world-space position where the ray hit a surface, listing 4.6 illustrates how this process is done.

```
1 float dist = RayTCurrent();
2 float3 ray_origin = WorldRayOrigin();
3 float3 ray_dir = WorldRayDirection();
4 float4 hit_position = float4(ray_origin + mul(ray_dir, dist), 1.0f);
```

Listing 4.6: Computing the world-space position where the ray hits the geometry

Having computed the world-space position where the ray hits a surface, it is then possible to compute the voxel cell from which we have to sample to obtain the surface normal at the surface point where the ray hit. This is done by multiplying the hit position by the voxel-space projection matrix. Having sampled the normal, we need to convert it from an unsigned float back to its signed version as previously explained, then we can compute the reflected ray direction by utilizing the *reflect()* intrinsic. Listing 4.7 ilustrates this process.

```
1 hit_pos = mul(voxel_space_matrix, hit_position);
2 hit_pos.rgb /= hit_pos.w;
3 int3 map_pos = int3(hit_pos.x - 1, hit_pos.y - 1, hit_pos.z - 1);
4 uint packed_normal = normal_map[map_pos];
5 float4 normal = UnpackFloat4(packed_normal)/255;
6 normal.rgb = normal.rgb * 2 - 1;
7 float3 new_dir = reflect(normalize(ray_dir), normal.rgb);
```

Listing 4.7: Computing direction in which the intersected ray is reflected

4.3.2.5 Radiance Calculation & Injection

Lastly, the irradiance of the surface position must be calculated and injected into the voxel grid, possibly also tracing secondary rays with the irradiance value as their color. At this point, most of the work that must be done is very simple considering it has already been explained how to obtain the voxel cell which must read from and written to and how to compute the normal. The only remaining work is to compute the diffuse irradiance; in this prototype, only diffuse irradiance is computed, however, there is nothing that stops specular irradiance from being computed, this is merely a time-saving measure. compute diffuse irradiance merely involves multiplying the incident radiance by the dot product between the light ray and surface normal. After the dot product is computed, lighting is attenuated by diving by the distance traveled by the light ray squared and this value is multiplied by the dot product and the original incident radiance. With the irradiance having been computed, it can be injected into the voxel grid by utilizing the atomic moving average function that has been previously explained.

Tracing secondary rays happens the same way as primary rays, the only difference is that their origin, direction, and the color is based on the payload data and irradiance calculated from the primary ray. This process can be repeated for as many indirect bounces as desired, and the main bottleneck will be the GPU's capabilities and not the versatility of the technique.

One final consideration is that this process could have been done inside the closest-hit shader. The difference in terms of final performance between running this process in the raygen shader and running it in the closest-hit shader has not been explored.

4.4 Final Rendering Stage

This section goes over the final implementation details about how to bring the data from the previous stages together to produce the final image.

4.4.1 Sampling Irradiance

The position from which the irradiance grid must be sampled is computed just as explained before, by using the voxel-space projection matrix. Then, by using linear sampling we obtain the value for each pixel that can be used to compute the final pixel color. One detail about this step, is that to combat an issue that will be explored in chapter 5, 7 samples must be taken and averaged instead of merely taking one. These 7 samples are

taken from the original voxel cell and from the 6 cells immediately surrounding it. Listing 4.8 illustrates this process.

```
float4 sample_voxel_grid(Texture3D texture, float3 voxel_position, float
    grid_resolution)
{
    float4 sample_sum = float4(0, 0, 0, 1);

    uint sample_count = 7;

    float4 samples[7];
    samples[0] = texture.Sample(gsampler, float3( voxel_position /
        grid_resolution)));
    samples[1] = texture.Sample(gsampler, float3((voxel_position + float3(
        1.0, 0, 0)) / grid_resolution));
    samples[2] = texture.Sample(gsampler, float3((voxel_position + float3
        (-1.0, 0, 0)) / grid_resolution));
    samples[3] = texture.Sample(gsampler, float3((voxel_position + float3(
        0, 1.0, 0)) / grid_resolution));
    samples[4] = texture.Sample(gsampler, float3((voxel_position + float3(
        0,-1.0, 0)) / grid_resolution));
    samples[5] = texture.Sample(gsampler, float3((voxel_position + float3(
        0, 0, 1.0)) / grid_resolution));
    samples[6] = texture.Sample(gsampler, float3((voxel_position + float3(
        0, 0,-1.0)) / grid_resolution));

  [unroll]
    for (uint i = 0; i < sample_count;++i)
    {
        sample_sum.rgb += samples[i].rgb;
    }

    float4 result = float4(sample_sum.rgb / sample_count, 1.0f);

    return result;
}
```

Listing 4.8: Sampling the irradiance grid with 7 samples

4.4.2 Final Pixel Value

The final pixel value is very simple to calculate, it merely involves multiplying the irradiance value by the diffuse coefficient, which in most cases will be a texel from a texture. Optionally, ambient illumination can be added, in this work, an ambient illumination that corresponds to 0.2 of the original diffuse texture is added. Listing 4.9 demonstrates how the final pixel value is computed.

```
1 float4 diffuse = diffuse_texture.Sample(linear_sampler, input.uv);
2 float4 irradiance = sample_voxel_grid(irradiance_grid, voxel_position,
      VOXEL_GRID_RESOLUTION);
3 float4 result = 0.2 * diffuse + irradiance * diffuse;
```

Listing 4.9: Computing the final color

Chapter 5

Result Analysis

The previous two chapters have gone over the theoretical and practical aspects used in the development process of this work. It is now important to analyze and validate the results that the implemented solution has been able to achieve. In that sense, this chapter will focus on two aspects, the produced image quality, and the runtime performance.

This chapter aims to analyze these two aspects of the solution implementation in various scenarios and configurations, as well as identify issues in the produced image quality and performance, in order to ascertain if those are an implementation issue or an issue with the solution architecture itself.

Additionally, the resulting image quality will be analyzed through a public user perception survey, that aimed to understand how possible users perceived the resulting image quality when compared to pre-rendered images.

5.1 Image Quality Analysis

Image quality is one of the goals of this project, as such this section focuses on that aspect. While image quality is important in the context of this project it must also be taken with a grain of salt, due to the usage of OBJ models, physically-based material models are not possible of being used and technical aspects that are outside of the scope of this project will also be limiting the attainable image quality, this section will also approach these limiting factors. Another reason why image quality is somewhat of a double edge sword during analysis is the fact that the desired image quality target will change with the project in question and its artistic direction, and since this technique is flexible enough to support any desired material-model it becomes hard to quantify the image quality reached in this project. For this reason, when analyzing image quality there will be an emphasis

on understanding, if a particular issue is caused by the solutions' methodology or its implementation, or even an unrelated shading issue.

Throughout this section, the Sponza scene will be the main scene used to evaluate image quality due to it not relying on textureless materials which are not supported properly at the moment. This affects the image quality of scenes likes Sibenik, however, since this is an implementation issue and not a methodological issue, this particular tradeoff is not considered problematic.

5.1.1 Achieved Result

Before moving into any detailed image quality analysis, it is important to establish a baseline of what has been achieved in terms of image quality. Figure 5.1 presents the Sponza scene rendered at a video output resolution of 1920x1080, a voxel grid of 128x128x128, and a ray-tracing resolution of 512x512. Figure 5.2 presents the various individual voxel grids that are used to construct the presented final image. It is important to note that while the diffuse coefficient voxel grid is presented here, secondary light rays are disabled, therefore its effect is not being seen as it is only used in secondary rays.

Figure 5.1: Still frame demonstrating the image quality achieved in this project.

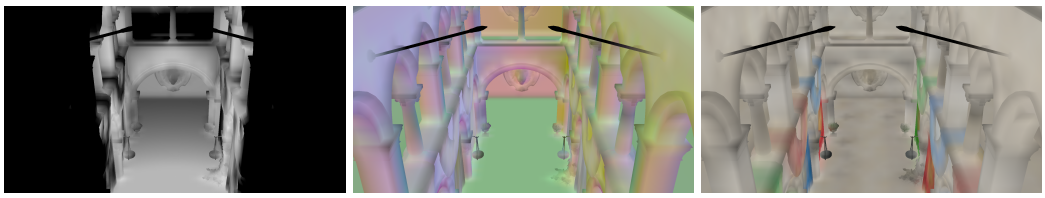

Figure 5.2: (Left) Irradiance Voxel Grid. (Middle) Surface Normal Voxel Grid. (Right) Diffuse Coefficient Voxel Grid

5.1.2 Structural Similarity Automatic Image Quality Analysis

An automated image quality analysis system was used to try and obtain a somewhat objective metric on the achieved image quality. This system utilizes the structural similarity index which takes into account the human visual perception system.

This section presents 5 sets of images corresponding to each of the test scenarios. In all of these sets of images generated in Blender, the left image is the image being tested, the middle image is the reference images and the rightmost image is the generated structural similarity index difference map. The caption of these images will indicate the score attributed to the reference image, which ranges from 0 to 1, indicating how similar to the reference image the image being tested is.

These metrics are great at giving us an instant idea of how close a real-time image compares to a pre-rendered one, however, this does not make it an absolute indicator of quality. Many aspects can reduce the score obtained that may not impact the actual decision of a human, a prime example of this in this work is the positioning of objects. As it was not possible to map the coordinates from the offline renderer to the developed renderer, therefore, a manual adjustment had to be made to map light positions and camera properties as closely as possible, unfortunately, this is error-prone. Another aspect to consider is that the way the scene is shaded will never be fully equal between the two renderers. Even if just due to light attenuations, this will also impact the overall scores. For these reasons, it was concluded that it would be a good idea to also include the structural similarity difference map in these figures.

The results of this analysis can be seen in figures 5.3, 5.4, 5.5, 5.6, and 5.7.

Figure 5.3: SSIM Score: 0.42831

Figure 5.4: SSIM Score: 0.53745

Figure 5.5: SSIM Score: 0.58612

Figure 5.6: SSIM Score: 0.3695

Figure 5.7: SSIM Score: 0.4861

5.1.3 Missing Data In Voxel Grid

During the solution implementation chapter, it was briefly mentioned that 7 samples were taken from the irradiance voxel grid and averaged together, however, the reasoning behind this was never explained. This choice was made as a workaround to a problem in the implementation of the voxelization process that will now be highlighted.

Figure 5.8 illustrates the problem that brought about the need for this workaround. When looking at the arches on the sides, a pattern of light streaks is visible. Upon closer inspection, it was discovered that this was caused by mixing linear filtering of the voxel grid with a section of the voxel grid that was missing data entirely. Figure 5.9 demonstrates this missing data in the various voxel grids that are used to render the final image.

Currently, the reason for this missing data is still unknown but it is known to be an implementation issue and not an issue with the solution architecture itself. Initially, this was speculated to be an issue with rays not hitting the particular voxels, however, after analyzing point sampled versions of the voxel grids, this option was fully discarded. This is the reason why a workaround that uses 7 samples was used instead of solving the root problem, however, it is notable that this approach only mitigates the issue and does not fully eliminate it.

Figure 5.8: Rendered image using a single sample per fragment from the radiance voxel grid. Arches contain visible unnatural light streak patterns.

Figure 5.9: Point sampled voxel grids with missing data at the same position. (Left) Surface normals. (Middle)Irradiance. (Right) Diffuse Coefficients.

5.1.4 Texture Aliasing

Texture aliasing is one factor that is outside of the scope of this project, which is limiting its image quality. Texture aliasing refers to patterns appearing in textures as the distance to the texture increases. Figure 5.10 illustrates this problem, where the left image has no aliasing but in the right one these oval patterns start appearing.

Texture aliasing is a result of texture minification and is particularly noticeable in high-frequency textures. A very simplified explanation of the cause for this problem is that as the polygon containing the texture becomes smaller as fewer and fewer fragments will be generated and texels will be skipped leading to these patterns. The only way to solve this issue is to utilize texture mipmapping, however, since that feature is not present in the developed renderer and due to time constraints texture aliasing was not addressed.

Figure 5.10: Demonstrating texture aliasing, the top image has no aliasing while in the bottom image, patterns are visible.

5.1.5 Lack of Anisotropic Voxelization

As previously explained, the usage of anisotropic voxelization is very important to obtain accurate results, however, due to time constraints this was not implemented, meaning that this is a limiting factor of the generated image quality. This section will explore the effect that the lack of anisotropic voxelization has on the generated images.

In the Sponza scene, the lack of anisotropic voxelization is particularly noticeable in thin geometry, such as the flags and curtains, making is susceptible to nullifying surface normals in the averaging process or of having nearby geometry data blended into its voxel.

Figure 5.11 shows a visible scenario where lighting is incorrect due to the lack of anisotropic voxelization.

Figure 5.11: Situation where lack of anisotropic voxelization lowers quality. In the left image, the irradiance grid is linearly sampled, while in the right image the grid is point sampled.

5.1.6 User Survey

As previously stated, a user survey was conducted via "Google Forms" which aimed to understand how real people perceived the images generated by this work. The survey was divided into two major sections: a head-to-head comparison and an individual image section. In the head-to-head section, the images generated by this work were pitted against pre-rendered images and users had to select which they thought looked better and give a brief explanation to their decision. In the individual image section, users were presented only with images generated by this work and were asked if they perceived any visual artifacts and to describe them.

The test images used were the same as those used in section 5.1.2 and exactly in the same order. Henceforth these will be referred to as Sibenik 1, Sibenik 2, Sibenik 3, Sponza 1, and Sponza 2 respectively. As of writing this document, the total number of answers to this survey was 222 and all answers can be viewed using this link.

5.1.6.1 Sibenik 1

For this test case, 92.8% of users preferred the pre-rendered image and pointed to visual artifacts such as the lack of contact shadows, a general sense of blur throughout the scene, the door lacking detail, and noise in the furthest walls. Additionally, 78.4% said they could perceive artifacts in the images generated by this work.

Lack of contact shadows is, unfortunately, an open problem for this work. Due to the need for texture filtering all generated shadows are soft. Blur seems to be related to loss of geometrical detail compared to the pre-rendered version. The reason for this is currently unknown as, there is no special processing done to the data outputted by the tinyobjloader model loader, although one possible reason is the lack of tesselation. The

door detail issues come from it being a textureless material which the current shader setup does not handle properly. Finally, the noise in the wall is an issue with texture sampling, possibly the lack of texture mipmapping.

Most of the issues presented seem to be implementation issues, except for the lack of contact shadows which is indeed an issue with the solution itself.

5.1.6.2 Sibenik 2

In this scenario, 75.2% prefer the pre-rendered image and pointed to many of the same issues as in Sibenik 1, except for the addition of the lighting discontinuity in the pillar. Additionally, 41.4% said they could perceive artifacts in the images generated by this work.

Regarding the lighting discontinuity in the pillar, this is assumed to be a result of the lack of anisotropic voxelization.

5.1.6.3 Sibenik 3

In this case, 97.7% preferred the pre-rendered image, pointing to the same issues as in sibenik 1. 23.9% said they could perceive visual artifacts.

5.1.6.4 Sponza 1

In this scenario, 46.4% preferred the pre-rendered image, where the main reported issue being the lack of self-shadowing on the lion head and the lighting discontinuities in the flags and curtains. Additionally, 76.1% said they could see artifacts in the images generated by this work.

The issue regarding flags and curtains has been explained in the section about the lack of anisotropic voxelization. The absence of self-shadowing in the lion head initially is due to not using normal-mapping, however, even if it was used it is unlikely the result would have been pleasing or even visible due to multiple fragments being averaged into the same voxel.

Once again shadowing, in general, is an unanswered question by the solution architecture.

5.1.6.5 Sponza 2

In this scenario, 51.8% preferred the pre-rendered image with the issues pointed out being the same as in other scenarios, which were already previously discussed. 28.4% reported being able to see artifacts.

5.2 Runtime Performance Analysis

The result of this project is meant to be applied in real-time scenarios, this means that analyzing its performance characteristics is fundamental to its real-world usage. This section focuses precisely on that aspect, going over how the technique performs under different scenarios and providing some conclusions on those results.

5.2.1 Benchmark Platform

When carrying out performance analysis, the external conditions under which it was conducted it was conducted are fundamental, as the results obtained are inherently tied to the hardware of the testing system. As such, in this section, the system configuration which was used to conduct this analysis is specified:

- CPU: AMD Ryzen 7 3700X
- GPU: Nvidia RTX 2070 Super
- RAM: 16GB 3200MHz
- Operating System: Windows 10 Pro 1909 Build 18363.900
- GPU Driver: Nvidia Gameready Driver Version 446.14
- C++ Compiler: MSVC Version 19.26.28806
- C++ Linker: MSVC Version 14.26.28806.0

5.2.2 Base Benchmark

Establishing how the developed behaves in different scenes under the same quality configuration is an important part to be able to analyze all further results and tests under the correct perspective. For these tests the voxel grid resolution was 128x128x128, the ray-tracing resolution was 512x512, the video output resolution was 1920x1080 and all textures(including the voxel grid) are sampled with linear filtering. Table 5.1 establishes the results of the particular test.

From these results, it can be concluded that the runtime cost of the technique scales with world-space complexity, which is primarily due to the need to issue a significant number of draw calls for each different material in the scene. Furthermore, the ray-tracing step also becomes more expensive with the increase of world-space complexity.

	Voxelization Stage (ms)	Radiance Injection Stage (ms)	Final Pass (ms)	Total Rendering Time (ms)
Sibenik Chathedral	0.891594	0.708464	0.147434	1.74749
Critek Sponza	2.24453	1.67343	0.466192	4.38415

Table 5.1: Benchmark establishing performance for two different scenes at a "recommended" configuration

An increase in the radiance injection stage time was initially not expected, however, upon further analysis this seems to be due to the voxel-ray overload problem. During the testing phase, it was concluded that Crytek's Sponza is particularly susceptible to this issue when compared to Sibenik.

A second easy conclusion that can be taken from this test is that Sponza is significantly more expensive to render than Sibenik, because Sponza uses a significantly higher number of materials. After further analysis, it was discovered that Sibenik takes 15 draw calls to be rendered in the final rendering pass, while sponza takes 25. If these values are added to the number of draw calls needed to voxelize the scene and the ray-tracing dispatch, Sibenik uses a total of 61 draw calls in a single frame while Sponza requires 101, therefore, justifying the generally higher cost of rendering the Sponza scene.

5.2.3 Video Output Resolution Benchmark

The key research question behind this work is whether or not a voxel strategy could be used to decouple ray-tracing resolution from video output resolution. For this reason, this is one of the most important aspects of this chapter and that on which this section focuses. Table 7.1 tests the hypothesis under several different video output resolution, voxel grid resolution, and ray-tracing resolution combinations. This particular test was conducted in the Sponza scene.

After performing this benchmark, it was concluded that the answer to the research question is yes: across all tested combinations the difference between video output resolution on the radiance injection stage and voxelization stage was negligible and inherently due to the randomness associated to performance testing. The final rendering pass is, however, affected by video output resolution. That being said, that is to be expected as that stage is responsible for outputting the video and will naturally need to shade more fragments the higher the video resolution. As such, we consider that this work achieved its goal of creating a technique that decouples ray-tracing resolution from video output resolution.

Nevertheless, while the ray-tracing resolution has been decoupled from video output resolution, a new coupling issue has appeared with the voxel grid resolution due to the voxel-ray overload problem. The lower the resolution of the voxel grid the more impact the radiance injection stage will have on rendering times, due to the number of rays hitting the same voxel.

5.2.4 Voxel Writing Strategy Benchmark

Up to this point the voxel-ray overload problem has been mentioned as a big issue of this technique. This section focuses on quantifying its impact. To test this, two scenarios were created, a favorable one where the impact of voxel-ray overloading was minimized while still producing meaningful results, by placing the light in an ample space not too close to any geometry, then an unfavorable scenario was created by placing the light as close as possible to flat geometry to maximize the spinlock runtime. These scenarios were created through empirically measuring performance and finding the place that best fits the descriptions of the scenarios.

This particular test was conducted in the Sibenik scene where it is much easier to differentiate between the favorable and unfavorable scenario, while Sponza makes it harder to do this due to its topology.

Three different tests were then conducted, one where atomic writing was not used displayed in table 7.2, one that used atomic averaging as mentioned in the solution architecture chapter demonstrated in table 7.3, and one where atomic maximum writes were used demonstrated in table 7.4.

When no atomic writes are used, the performance characteristics of the applications are quite favorable, with the voxelization stage always being by far the biggest bottleneck. This means that voxel-ray overloading has been completely removed, unfortunately, this approach cannot be used due to it not synchronizing writes into the voxel grid, leading to flickering. Since there is no voxel-ray overloading it is easy to surmise that there would be no significant difference between the favorable and unfavorable scenarios and the data acquired backs up this hypothesis.

When atomic averages are used, the impact of this approach when compared to not using atomic writes is immediately apparent, as even in the best possible scenarios, the performance is significantly worse. It is also easy to see that the radiance injection stage is not the only one affected by the increased runtime of the spinlock, seeing as the voxelization stage also suffers heavily from it. Lastly, it is also possible to observe that the lower the resolution of the voxel grid the more and more apparent the voxel-ray overload

becomes. This is because more rays are hitting the same voxel, causing the spinlock to run for longer.

When atomic maximum writes are used, the impact on performance is significantly lower than atomic averages. In favorable scenarios, the performance impact is almost non-existent when compared to not using atomic writes, whereas under unfavorable scenarios, while the impact is still visible, it is significantly lower than with atomic averages. A relationship between the grid resolution and the impact of the write strategy still exists, however, its impact is significantly lower.

It is possible to conclude that the ray-overload problem still exists since memory still has to be synced to execute atomic operations, however, it has been minimized to the point where it is almost unnoticeable. While the major beneficiary from this approach is the radiance injection stage, the voxelization stage also benefits significantly. The main detractor of this approach is that it is considered less accurate due to operating on the packed colors, although, considering the performance data here presented, this approach seems much saner to be used in real-world scenarios than atomic averaging, at least under the scenarios tested here.

5.2.5 Secondary Ray Benchmark

Up to this point, all tests have been run with secondary rays disabled to keep them focused on what is being tested, however, it is important to understand the impact that using secondary rays has. The test scenario for this section is very similar to those used in subsection 5.2.4, the only difference being that instead of testing writing strategy, the difference between using no secondary rays and using a single secondary ray is being analyzed. A single indirect ray is used due to its implementation simplicity, while still helping to identify the actual impact of secondary rays. Table 7.5 demonstrates the data obtained in this test when no secondary rays are used, while table 7.6 demonstrates the results obtained when a single secondary ray is used. This test was conducted in the Sibenik scene for the same reason that was stated in the atomic writing strategy section.

The main aspect that can be observed is that while secondary rays have a visible impact, their impact is not as significant as the primary rays despite the amount of primary and secondary rays being roughly the same. This implies that there is a cost to starting the ray tracing pipeline besides just executing the actual ray-tracing process.

The second noticeable characteristic is that when secondary rays are used, the impact of the voxel-ray overload problem in the most extreme cases is reduced. Currently, the reason for this is unknown with the only theory being that it forces the GPU to increase

its frequency due to the extra load. Considering that GPU's execute code in lockstep this is unlikely to be the case.

5.2.6 Sampling Strategy Benchmark

As previously explained, during the final rendering stage, 7 samples are taken from the radiance voxel grid and averaged to work around the missing data in the voxel grid. While this is not an issue with the methodology it is also important to understand how this particular workaround impacts performance. This section will explore this topic as well as an attempt to understand how the resolution of the voxel grid affects the performance of the final pass.

To test this, two scenarios were created, one where only a single sample is taken from the voxel grid and another where 7 are taken. These two scenarios are then explored at different video output and voxel grid resolutions with a constant ray tracing resolution. This test was conducted in the Sibenik scene. Table 7.7 illustrates the results obtained for this particular test.

The first conclusion obtained from this data is that higher voxel grid resolution does indeed have a higher cost, however, its impact is negligible. The reason for this cost may be assumed to originate from the increased amount of texture cache misses, More fragments will have different voxel grid coordinates s its resolution increases.. The second more significant conclusion is that the 7-sample strategy does indeed have a noticeable cost, which is negligible at lower video output resolutions, however, it is very significant at higher video output resolutions. This being said, this strategy is a workaround to a particular implementation issue and not a problem with the solution architecture itself, therefore it should not be viewed as a reason not to use the proposed solution.

Chapter 6

Conclusion

Having gone through a revision of the state of the art in real-time global illumination rendering, voxels were seen as a promising avenue to mix traditional rasterization and hardware-accelerated ray-tracing. As such a question was asked. Could a voxel approach be used to decouple the resolution at which rays must be traced from the output resolution of the render target?

Theoretically, everything pointed to the answer being yes. Having gone through the development of this work, the real-world answer is yes, under the conditions that this work was developed, however, questions still linger about the validity of this answer in other scenarios.

The solution that was reached appears to be versatile and flexible, not putting barriers to future extension and not hindering the art direction of projects in which it is used. That being said, this solution has only been explored in a very constrained scenario and with a limited implementation.

As the author of this work, and after having developed a prototype of this technique, I believe that it has significant potential. That being said, much work is still required for it to meet that potential, which goes significantly beyond the scope of the integrated master's dissertation it is inserted in. Pursuing future work related to this technique is likely not a fruitless endeavor, based on the results achieved both in this work and with similar techniques. Even now, new voxel-based techniques are still being developed to simulate global illumination in systems that do not support hardware-accelerated ray tracing. As such there is nothing that stands in the way of it also being used with actual ray-tracing.

6.1 Future Work

The solution presented, while flexible and versatile, leaves many aspects unexplored, and the same can be said for the solution's implementation which is still missing some of the core features from the solution architecture. This is preventing it from reaching its full potential. This chapter focuses on these aspects, the work that each of these two components still has ahead of them to be considered complete. As such, this chapter is broken down into future work regarding the solution architecture itself, and future work on its implementation. An important aspect to consider is that despite being divided, these two things are still intimately connected, with obstacles and problems during implementation many times driving changes to the architecture. As such, any of the ideas proposed as future work for the solution implementation is capable of creating new avenues to be explored and incorporated into the solution architecture.

6.1.1 Solution Architecture

The solution architecture leaves two main questions after having been implemented, one being that of the voxel-ray overload problem and the other being the issue of shadows.

The voxel-ray overload problem can be significantly mitigated through atomic max operations instead of atomic averaging, however, the price to pay is accuracy. A line of work that should be followed here is finding a solution that can bring these two approaches together, in order to mitigate the cost of atomic averaging while not completely giving up accuracy. This being said, the loss of accuracy of the atomic max operation is in itself a lingering question. Due to the various other aspects limiting the rendering quality of the implementation, it is hard to tell if the loss of accuracy caused by the atomic max operations is indeed significant. As such, before searching for a new approach to atomic writes into the voxel grid, it would be wise to better understand how much atomic maxima operations degrade quality when compared to atomic averages.

Regarding shadows, the main problem faced by the solution is that, due to the need for linear filtering, there is always a gradient between lit and unlit areas of the scene, making contact shadows non-existent. As such, providing a solution to rendering more accurate shadows is an important line of future work. Close-up shadows need to be sharper, and even in the case of soft-shadows, which currently do look realistic, there are possible improvements to be made regarding their attenuation. A possible line of work is trying to retrofit existing shadowing techniques into this technique to fill in the gaps.

These two questions are the clearest ones that one could reach after reading this document, however, there are some more lines of work that could be explored. One that

was briefly mentioned is that of the ray-tracing acceleration structure. This is an intricate topic that was not explored in this work though it could have an important impact on the resulting performance of the technique. As such, conducting further research into the acceleration structure is a viable line of future work.

Lastly, there is the topic of transparencies and translucent materials, which were briefly mentioned but not further explored. Exploring ways of modeling light interactions with these types of materials under the framework proposed in this project would be a promising line of future work.

6.1.2 Solution Implementation

Future work to the solution implementation clearly would start at fixing the current issues it has, starting with the issues to image quality.

- Anisotropic voxelization needs to be added to reduce light leaking and issues related to thin geometry.
- Texture mipmapping should be added to reduced texture aliasing.
- Normal mapping support to obtain a higher amount of detail for the same geometry and have more accurate normals in general.
- Improving the way secondary rays are traced, using multiple diffuse rays instead of a single ray reflected along the normal.
- Support for physically based material models. And last but not least the problem with missing information in the voxel grid.

These are the points that focus on fixing implementation issues, but there are also more exploratory and speculative aspects. Currently, the implementation utilizes light-based ray-tracing, however, exploring the impact and interactions of camera-based ray-tracing is a highly important line of future work. Exploring the usage of stochastic ray-tracing and what implications this would have on the solution architecture is also a viable line of future work.

Then there are aspects that would be considered optimizations to the implementation, such as the implementation of a clipmap based voxel grid and swapping voxelization over to a single render pass strategy. Additionally, this work has not explored techniques for GPU shader optimization due to the complexity of this topic alone, therefore, spending time to optimize the shaders used for the various states is a crucial line of future work.

Overall, the solution implementation has much that can be seen as future work, and certainly, some of the more exploratory lines of future work will lead to a need to readjust the solution architecture. This being said, considering the time frame as well as the scope of this dissertation and due to leveraging state-of-the-art technology, this work can in many ways be seen as a complete success. After all, being open to improvement is not something negative, but rather something that should be viewed as positive for the field.

Chapter 7

Annexes

This chapter serves merely to contain code annexes that might be relevant to the reader, but would otherwise break the overall chapter flow if added as listing to the respective chapter. This chapter will, therefore, be broken down into sections corresponding to the chapter they originate from.

7.1 Solution Implementation Annexes

```
ComPtr<ID3D12Resource> voxel_grid_volume = nullptr;

D3D12_RESOURCE_DESC resource_desc;
resource_desc.Alignment = 0;
resource_desc.Width = VOXEL_GRID_WIDTH;
resource_desc.Height = VOXEL_GRID_HEIGH;
resource_desc.DepthOrArraySize = VOXEL_GRID_DEPTH;
resource_desc.MipLevels = 1;
resource_desc.SampleDesc.Count = 1;
resource_desc.SampleDesc.Quality = 0;
resource_desc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
resource_desc.Flags = D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS;
resource_desc.Format = VOXEL_GRID_FORMAT_TYPELESS;
resource_desc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE3D;

D3D12_HEAP_PROPERTIES heap_properties = {};.
heap_properties.Type = D3D12_HEAP_TYPE_DEFAULT;
heap_properties.CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_UNKNOWN;
heap_properties.MemoryPoolPreference = D3D12_MEMORY_POOL_UNKNOWN;
heap_properties.CreationNodeMask = 1;
```

```
heap_properties.VisibleNodeMask = 1;

device->CreateCommittedResource(&heap_properties,
                                D3D12_HEAP_FLAG_NONE,
                                &resource_desc,
                                D3D12_RESOURCE_STATE_UNORDERED_ACCESS,
                                nullptr,
                                IID_PPV_ARGS(&voxel_grid_volume));
```

Listing 7.1: Creating A Voxel Grid Volume Resource in D3D12

```

UINT srv_heap_index = SRHeapManager::GetManager().Allocate();
DescriptorHeap descriptor_heap = SRHeapManager::GetManager().
    descriptor_heap;

D3D12_SHADER_RESOURCE_VIEW_DESC srv_desc = {};
srv_desc.Format = VOXEL_GRID_FLOAT_SRV_FORMAT;//Made up placeholder to the
     actual SRV format
srv_desc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE3D;
srv_desc.Shader4ComponentMapping =
    D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
srv_desc.Texture3D.MostDetailedMip = 0;
srv_desc.Texture3D.MipLevels = 1;
srv_desc.Texture3D.ResourceMinLODClamp = 0;

device->CreateShaderResourceView(voxel_grid_volume.Get(), &srv_desc, (
    descriptor_heap)[srv_heap_index]);
```

Listing 7.2: Creating a shader resource view into the voxel volume in D3D12

```

UINT heap_index = SRHeapManager::GetManager().Allocate();
DescriptorHeap descriptor_heap = SRHeapManager::GetManager().
    descriptor_heap;

D3D12_UNORDERED_ACCESS_VIEW_DESC uav_desc = {};
uav_desc.Format = VOXEL_GRID_INTEGER_UAV_FORMAT;//Made up placeholder to
    the actual UAV format
uav_desc.ViewDimension = D3D12_UAV_DIMENSION_TEXTURE3D;
uav_desc.Texture3D.MipSlice = 0;
uav_desc.Texture3D.FirstWSlice = 0;
```

```
uav_desc.Texture3D.WSize = depth;

device->CreateUnorderedAccessView(voxel_grid_volume.Get(),
                                  nullptr,
                                  &uav_desc,
                                  (descriptor_heap)[heap_index]);
device->CreateUnorderedAccessView(voxel_grid_volume.Get(),
                                  nullptr,
                                  &uav_desc,
                                  clear_heap[0]);
```

Listing 7.3: Creating an unordered access view into the voxel volume in D3D12

```

command_list->SetPipelineState(pso);
command_list.BindTexture(material->texture,2,3);
command_list.BindIndexBuffer(index_buffer);
command_list.BindConstantBuffer(material->material_cbuffer,4);
//Z-Axis View
ZAxisVoxelizationCall(command_list, 0);
//X-Axis View
XAxisVoxelizationCall(command_list, 0);
//Y-Axis View
YAxisVoxelizationCall(command_list, 0);
```

Listing 7.4: Voxelizing a set of objects with the same material

```

UpdateVoxelizationMatrices(XMMatrixRotationAxis({0.0f,1.0f,0.0f},0.5f *
    M_PI));
UpdateVoxelizationCBufferX();
command_list.BindConstantBuffer(voxelization_cbuffer_x,
    constant_buffer_slot);
command_list.SendDrawCall();
```

Listing 7.5: Functions that issues a single voxelization draw call for the x-axis

```

cbuffer ConversionMatrixBuffer : register(b0)
{
    float4x4 ClipSpaceMatrix;
```

```
5       float4x4 VoxelSpaceMatrix;
6   };
7
8   cbuffer MaterialProperties : register(b1)
9   {
10      float4 ambient_coefficient;
11      float4 diffuse_coefficient;
12      float4 specular_coefficient;
13      float4 specular_exponent;
14  }
15
16  RWTexture3D<uint> albedo_map : register(u0);
17  RWTexture3D<uint> ocupancy_map : register(u1);
18  RWTexture3D<uint> diffuse_map : register(u2);
19  RWTexture3D<uint> specular_map : register(u3);
20  RWTexture3D<uint> exponent_map : register(u4);
21  RWTexture3D<uint> normal_map : register(u5);
22
23  Texture2D gText : register(t0);
24  SamplerState gsampler : register(s0);
25
26  struct VS_OUTPUT
27  {
28      float4 position : SV_POSITION;
29      float4 voxel_grip_position : VOXEL_POS;
30      float4 normal : NORMAL;
31      float2 uv : UV;
32  };
33
34  struct VS_INPUT
35  {
36      float3 pos : POSITION;
37      float2 uv : UV;
38      float3 normal : NORMAL;
39      float3 color : COLOR;
40  };
41
42  VS_OUTPUT VoxelVSMain(VS_INPUT input)
43  {
44      VS_OUTPUT output;
45
46      output.position = mul(ClipSpaceMatrix, float4(input.pos, 1.0f));
47      output.voxel_grip_position = mul(VoxelSpaceMatrix, float4(input.pos,
            1.0f));
```

```
48      output.uv = input.uv;
49      output.normal = float4(normalize(input.normal) * 0.5 + 0.5, 1.0f);
50
51      return output;
52  }
53
54  struct PS_OUTPUT
55  {
56      float4 color : SV_TARGET;
57  };
58
59  PS_OUTPUT VoxelPSMain(VS_OUTPUT input)
60  {
61      PS_OUTPUT output;
62
63      float3 gridPos = input.voxel_grip_position.xyz / input.
            voxel_grip_position.w;
64      int3 voxel_pos = int3(gridPos.x - 1, gridPos.y - 1, gridPos.z - 1);
65
66      ocupancy_map[voxel_pos] = uint(1);
67      float4 frag_color = gText.Sample(gsampler, input.uv);
68      AverageRGBA8Voxel(albedo_map, voxel_pos, frag_color);
69      AverageRGBA8Voxel(diffuse_map, voxel_pos, float4(diffuse_coefficient.
            rgb,1));
70      AverageRGBA8Voxel(specular_map, voxel_pos, specular_coefficient);
71      AverageRGBA8Voxel(normal_map, voxel_pos, input.normal);
72      output.color = input.voxel_grip_position / 256.0f;
73      discard;
74
75      return output;
76  }
```

Listing 7.6: Complete voxelization shader used during voxelization draw calls. The vertex shader and pixel shader are merged into a single listing

```
1  #define NV_SHADER_EXTN_SLOT u10
2  #define NV_SHADER_EXTN_REGISTER_SPACE space0
3  #include "ThirdParty/nvapi/nvHLSLExtns.h"
4
5  struct BuiltinIntersectionAttribs
6  { // Barycentric coordinates of hit in
7      float2 barycentrics; // the triangle are: (1-x-y, x, y)
8  };
```

```
9
10 cbuffer RTCBuffer : register(b0)
11 {
12     float4x4 voxel_space_matrix;
13     float4 light_color;
14     float3 light_position;
15     float light_radius;
16     float light_extent;
17 };
18
19 RaytracingAccelerationStructure Scene : register(t1);
20 RWTexture3D<uint> RenderTarget : register(u0);
21 RWTexture3D<uint> normal_map : register(u1);
22
23 // Packs float4 in [0,1] range into [0-255] uint
24 uint PackFloat4(float4 val)
25 {
26     uint r = round(clamp(val.r, 0.0, 1.0) * 255.0);
27     uint g = round(clamp(val.g, 0.0, 1.0) * 255.0);
28     uint b = round(clamp(val.b, 0.0, 1.0) * 255.0);
29     uint a = round(clamp(val.a, 0.0, 1.0) * 255.0);
30     return (
31             (uint(a) & 0x000000FF) << 24U |
32             (uint(b) & 0x000000FF) << 16U |
33             (uint(g) & 0x000000FF) << 8U |
34             (uint(r) & 0x000000FF));
35 }
36
37 // Unpacks values and returns float4 in [0,1] range
38 float4 UnpackFloat4(uint val)
39 {
40     uint r = (val & 0x000000FF);
41     uint g = (val & 0x0000FF00) >> 8U;
42     uint b = (val & 0x00FF0000) >> 16U;
43     uint a = (val & 0xFF000000) >> 24U;
44     return float4(r / 255.0, g / 255.0, b / 255.0, a / 255.0);
45 }
46
47 void AverageRGBA8Voxel(RWTexture3D<uint> voxel_map, int3 voxel_coords,
    float4 val)
48 {
49     uint packed_color = PackFloat4(float4(val.rgb, 1.0/255.0));
50
51     uint previousStoredValue = 0;
```

```
52      uint currentStoredValue;
53
54      float4 currValue;
55      float3 average;
56      uint count;
57
58      InterlockedCompareExchange(voxel_map[voxel_coords],
            previousStoredValue, packed_color, currentStoredValue);
59      while (currentStoredValue != previousStoredValue)
60      {
61          previousStoredValue = currentStoredValue;
62          currValue = UnpackFloat4(previousStoredValue);
63
64          average = currValue.rgb;
65          count = round(currValue.a * 255.0);
66
67          average = (average * count + val.rgb) / (count + 1);
68
69          packed_color = PackFloat4(float4(average, (count + 1)/255.0));
70          InterlockedCompareExchange(voxel_map[voxel_coords],
                previousStoredValue, packed_color, currentStoredValue);
71      }
72  }
73
74  struct RayPayload
75  {
76      float4 color;
77      float3 origin;
78      float3 direction;
79      float distance;
80      bool missed;
81  };
82
83  [shader("raygeneration")]
84  void raygen()
85  {
86      // Initialize the ray payload
87      RayPayload payload;
88      payload.color = light_color;
89      payload.origin = float3(0, 0, 0);
90      payload.direction = float3(0, 0, 0);
91      payload.missed = false;
92      payload.distance = 0;
93
```

```
94      // Get the location within the dispatched 2D grid of work items
95      // (often maps to pixels, so this could represent a pixel coordinate).
96      uint2 launchIndex = DispatchRaysIndex().xy;
97      float2 dims = float2(DispatchRaysDimensions().xy);
98      float2 d = (((launchIndex.xy + 0.5f) / dims.xy) * 2.f - 1.f);
99
100     float rt_res = 512;
101     float3 direction = float3(DispatchRaysIndex().xyz) - float3(rt_res /
            2, rt_res / 2, rt_res / 2);
102     direction = normalize(direction);
103     float4 origin = mul(voxel_space_matrix, float4(light_position, 1.0f));
104
105     //ray.Origin = (origin.xyz / origin.w) - float3(1, 1, 1);
106     RayDesc ray;
107     ray.Origin = light_position;
108     ray.Direction = direction;
109     ray.TMin = 0.02;
110     ray.TMax = 100000;
111
112     //uint startTime = NvGetSpecial(9);
113     TraceRay(Scene, RAY_FLAG_FORCE_OPAQUE, 0xFF, 0, 0, 0, ray, payload);
114     //uint endTime = NvGetSpecial(9);
115     //uint deltaTime = endTime - startTime;
116
117     float light_intensity = 100.0f;
118     float4 irradiance_result = float4(0, 0, 0, 0);
119     if(!payload.missed)
120     {
121         //Shade Primary Hit
122         float4 hit_pos = float4(payload.origin, 1.0f);
123
124         hit_pos = mul(voxel_space_matrix, hit_pos);
125         hit_pos.rgb /= hit_pos.w;
126         int3 map_pos = int3(hit_pos.x - 1, hit_pos.y - 1, hit_pos.z - 1);
127
128         uint packed_normal = normal_map[map_pos];
129
130         float4 normal = UnpackFloat4(packed_normal);
131         normal.rgb = (normal.rgb * 2) - 1;
132
133         float NdotL = saturate(-dot(normalize(direction), normal.rgb));
134
135         float t = abs(payload.distance);
136         float falloff = (1.0 / abs(t*t));
```

```
137
138         float3 irradiance = ((light_intensity * NdotL * falloff) *
                light_color.rgb);
139         irradiance_result = float4(irradiance, 1.0f);
140
141         float3 value = normalize(payload.origin) * 0.5 + 0.5;
142
143         AverageRGBA8Voxel(RenderTarget, map_pos, float4((value), 1.0));
144     }
145
146     if(!payload.missed && false)
147     {
148         float3 value = normalize(payload.origin) * 0.5 + 0.5;
149
150         float3 reflection_direction = payload.direction;
151         payload.color = irradiance_result;
152         payload.origin = float3(0, 0, 0);
153         payload.direction = float3(0, 0, 0);
154         payload.missed = false;
155         payload.distance = 0;
156
157         ray.Origin = payload.origin;
158         ray.Direction = reflection_direction;
159
160
161         light_intensity = 1;
162
163         TraceRay(Scene, RAY_FLAG_FORCE_OPAQUE, 0xFF, 0, 0, 0, ray, payload
                );
164         {
165             //Shade Secondary Hit
166             float4 hit_pos = float4(payload.origin, 1.0f);
167
168             hit_pos = mul(voxel_space_matrix, hit_pos);
169             hit_pos.rgb /= hit_pos.w;
170             int3 map_pos = int3(hit_pos.x - 1, hit_pos.y - 1, hit_pos.z -
                    1);
171
172             uint packed_normal = normal_map[map_pos];
173
174             float4 normal = UnpackFloat4(packed_normal);
175             normal.rgb = (normal.rgb * 2) - 1;
176
```

```
177             float NdotL = saturate(-dot(normalize(reflection_direction),
                    normal.rgb));
178 
179             float t = abs(payload.distance);
180             float falloff = (1.0 / abs(t * t));
181 
182             float3 irradiance = ((light_intensity * NdotL * falloff) *
                    irradiance_result);
183 
184             AverageRGBA8Voxel(RenderTarget, map_pos, float4((value), 1.0))
                    ;
185         }
186     }
187 }
188 
189 [shader("intersection")]
190 void intersection()
191 {
192 }
193 
194 [shader("miss")]
195 void miss(inout RayPayload data : SV_RayPayload)
196 {
197     data.missed = true;
198 }
199 
200 [shader("anyhit")]
201 void anyhit(inout RayPayload data, BuiltinIntersectionAttribs hit)
202 {
203     data.color = float4(0.0f, 0.0f, 0.0f, 0.0f);
204 }
205 
206 [shader("closesthit")]
207 void closesthit(inout RayPayload data, in BuiltinIntersectionAttribs hit)
208 {
209     float dist = RayTCurrent();
210     float3 ray_origin = WorldRayOrigin();
211     float3 ray_dir = WorldRayDirection();
212 
213     float4 hit_position = float4(ray_origin + mul(ray_dir, dist), 1.0f);
214     float4 hit_pos = hit_position;
215     hit_pos = mul(voxel_space_matrix, hit_pos);
216     hit_pos.rgb /= hit_pos.w;
217 
```

```
218     int3 map_pos = int3(hit_pos.x - 1, hit_pos.y - 1, hit_pos.z - 1);
219     uint packed_normal = normal_map[map_pos];
220     float4 normal = UnpackFloat4(packed_normal);
221     normal.rgb = (normal.rgb * 2) - 1;
222 
223     if(normal.r != 0 || normal.g != 0 || normal.b != 0)
224     {
225         float3 new_dir = reflect((ray_dir.rgb), normal.rgb);
226         data.missed = false;
227         data.direction = new_dir;
228     }else
229     {
230         data.missed = true;
231         data.direction = float3(0, 0, 0);
232     }
233     data.color = float4(1, 0, 0, 1);
234     data.origin = hit_position.rgb;
235     data.distance = dist;
236 }
```

Listing 7.7: Ray Tracing Shaders

7.2 Results Analysis Annexes

Output Resolution	Voxel Grid Resolution	Ray Tracing Resolution	Voxelization Stage (ms)	Radiance Injection Stage (ms)	Final Pass (ms)	Total Rendering Time (ms)
640x480	64^3	256^2	1.06693	0.701148	0.127261	1.89534
		512^2	1.06526	5.17492	0.128425	6.36861
		1024^2	1.0606	15.3444	0.13051	16.5355
		2048^2	1.06553	32.7374	0.129433	33.9324
	128^3	256^2	2.21332	0.216585	0.130326	2.56023
		512^2	2.23071	1.61867	0.129541	3.97892
		1024^2	2.20061	7.63267	0.132658	9.96594
		2048^2	2.23385	23.2973	0.132261	25.6634
	256^3	256^2	7.18004	0.0810194	0.135261	7.39632
		512^2	7.18224	0.533723	0.134297	7.85026
		1024^2	7.29559	4.18572	0.135656	11.617
		2048^2	7.05089	18.672	0.136993	25.8599
1280x720	64^3	256^2	1.09162	0.707802	0.256129	2.05556
		512^2	1.08653	5.31049	0.255658	6.65269
		1024^2	1.06981	14.8035	0.256135	16.1295
		2048^2	1.07852	32.9899	0.253383	34.3218
	128^3	256^2	2.23149	0.212716	0.260691	2.7049
		512^2	2.21081	1.6687	0.258507	4.13802
		1024^2	2.21267	7.68073	0.260027	10.1534
		2048^2	2.19242	23.5159	0.257481	25.9658
	256^3	256^2	7.20955	0.0873412	0.264852	7.56175
		512^2	7.26135	0.524618	0.264695	8.05066
		1024^2	7.37712	4.19439	0.265016	11.8365
		2048^2	6.93482	18.7194	0.265526	25.9198
1920x1080	64^3	256^2	1.09425	0.708497	0.46785	2.2706
		512^2	1.11148	5.33797	0.466288	6.91574
		1024^2	1.13564	15.3877	0.470762	16.9941
		2048^2	1.12873	32.8909	0.468602	34.4882
	128^3	256^2	2.24251	0.212497	0.470922	2.92593
		512^2	1.66984	2.24154	0.469997	4.38138
		1024^2	2.24309	7.71947	0.467548	10.4301
		2048^2	2.22373	23.4691	0.471707	26.1645
	256^3	256^2	7.24855	0.0830808	0.477017	7.80865
		512^2	7.24524	0.588229	0.479569	8.31304
		1024^2	7.88697	4.30354	0.473531	12.664
		2048^2	7.12218	19.4013	0.477973	27.0014

Table 7.1: Benchmark data used to understand the impact of video output resolution on rendering times

Light Position	Voxel Grid Resolution	Ray Tracing Resolution	Voxelization Stage (ms)	Radiance Injection Stage (ms)	Final Pass (ms)	Total Rendering Time (ms)
Favorable Scenario	64^3	256^2	0.122103	0.0357491	0.155403	0.313255
		512^2	0.122231	0.0841889	0.155049	0.361468
		1024^2	0.12269	0.276428	0.155856	0.554973
		2048^2	0.121912	1.05602	0.155381	1.33332
	128^3	256^2	0.278515	0.0359526	0.158027	0.472494
		512^2	0.277599	0.0837135	0.157659	0.518972
		1024^2	0.277601	0.275462	0.158185	0.711248
		2048^2	0.277694	1.05526	0.158526	1.49148
	256^3	256^2	1.33941	0.0364394	0.163285	1.53913
		512^2	1.34545	0.0843984	0.164421	1.59426
		1024^2	1.34174	0.276209	0.163584	1.78153
		2048^2	1.34384	1.0669	0.163923	2.57466
Unfavorable Scenario	64^3	256^2	0.1216	0.0337566	0.154449	0.309806
		512^2	0.121817	0.0824932	0.154202	0.358512
		1024^2	0.121991	0.27721	0.154507	0.553708
		2048^2	0.122326	1.07471	0.155806	1.35284
	128^3	256^2	0.27854	0.0339983	0.156887	0.469426
		512^2	0.278221	0.0824997	0.15594	0.516661
		1024^2	0.278464	0.277051	0.156235	0.711749
		2048^2	0.279031	1.07486	0.156958	1.51085
	256^3	256^2	1.33588	0.0339783	0.162211	1.53207
		512^2	1.33476	0.0825003	0.161984	1.57925
		1024^2	1.33405	0.277543	0.162154	1.77374
		2048^2	1.33454	1.08126	0.162199	2.57799

Table 7.2: Technique performance when atomic writes are not utilized

Light Position	Voxel Grid Resolution	Ray Tracing Resolution	Voxelization Stage (ms)	Radiance Injection Stage (ms)	Final Pass (ms)	Total Rendering Time (ms)
Favorable Scenario	64^3	256^2	0.329934	0.203306	0.155127	0.688367
		512^2	0.329469	1.96377	0.154829	2.44807
		1024^2	0.329924	10.8743	0.155166	11.3594
		2048^2	0.326752	36.0532	0.154637	36.5346
	128^3	256^2	0.895788	0.0919789	0.157073	1.14484
		512^2	0.895832	0.712392	0.157881	1.7661
		1024^2	0.899321	7.12702	0.158475	8.18482
		2048^2	0.899012	30.5497	0.159022	31.6078
	256^3	256^2	2.95473	0.0514137	0.163731	3.16987
		512^2	2.95442	0.231721	0.164033	3.35017
		1024^2	2.9672	2.28362	0.164222	5.41504
		2048^2	2.95123	14.7621	0.164947	17.8783
Unfavorable Scenario	64^3	256^2	0.331321	1.34332	0.155385	1.83003
		512^2	0.331098	7.32731	0.156015	7.81442
		1024^2	0.329006	38.9989	0.156091	39.484
		2048^2	0.321996	164.36	0.158151	164.84
	128^3	256^2	0.897908	0.671663	0.157064	1.72663
		512^2	0.897689	11.3135	0.157457	12.3687
		1024^2	0.897679	12.5766	0.158837	13.6331
		2048^2	0.896514	49.2111	0.158721	50.2663
	256^3	256^2	2.94273	0.244941	0.16282	3.3505
		512^2	2.97297	2.77733	0.163289	5.91359
		1024^2	2.97265	21.8101	0.163835	24.9466
		2048^2	2.9606	42.8114	0.162547	45.9346

Table 7.3: Technique performance when using atomic averaging

Light Position	Voxel Grid Resolution	Ray Tracing Resolution	Voxelization Stage (ms)	Radiance Injection Stage (ms)	Final Pass (ms)	Total Rendering Time (ms)
Favorable Scenario	64^3	256^2	0.122102	0.0363803	0.156445	0.314927
		512^2	0.122046	0.0857298	0.156121	0.363896
		1024^2	0.122195	0.280066	0.155984	0.558245
		2048^2	0.122163	1.09329	0.15576	1.37121
	128^3	256^2	0.286022	0.0369032	0.158743	0.481668
		512^2	0.286838	0.0857186	0.15765	0.530207
		1024^2	0.285927	0.278029	0.158497	0.722452
		2048^2	0.286302	1.06002	0.157734	1.50405
	256^3	256^2	1.3801	0.0371671	0.164184	1.58145
		512^2	1.37667	0.0853268	0.163873	1.62587
		1024^2	1.37816	0.27794	0.163551	1.81965
		2048^2	1.38419	1.06926	0.163978	2.61743
Unfavorable Scenario	64^3	256^2	0.121888	0.0411665	0.154753	0.317808
		512^2	0.121922	0.144347	0.154309	0.420578
		1024^2	0.121844	0.560793	0.155224	0.837862
		2048^2	0.122026	2.24368	0.155427	2.52113
	128^3	256^2	0.285336	0.037234	0.157468	0.480038
		512^2	0.285406	0.125856	0.156043	0.567305
		1024^2	0.28616	0.521211	0.157257	0.964628
		2048^2	0.285223	2.12585	0.157417	2.56849
	256^3	256^2	1.38025	0.0347238	0.162754	1.57773
		512^2	1.38585	0.0915126	0.162723	1.64009
		1024^2	1.37613	0.39928	0.162696	1.93811
		2048^2	1.37441	1.69001	0.162943	3.22737

Table 7.4: Technique performance when using atomic maxima

Light Position	Voxel Grid Resolution	Ray Tracing Resolution	Voxelization Stage (ms)	Radiance Injection Stage (ms)	Final Pass (ms)	Total Rendering Time (ms)
Favorable Scenario	64^3	256^2	0.329934	0.203306	0.155127	0.688367
		512^2	0.329469	1.96377	0.154829	2.44807
		1024^2	0.329924	10.8743	0.155166	11.3594
		2048^2	0.326752	36.0532	0.154637	36.5346
	128^3	256^2	0.895788	0.0919789	0.157073	1.14484
		512^2	0.895832	0.712392	0.157881	1.7661
		1024^2	0.899321	7.12702	0.158475	8.18482
		2048^2	0.899012	30.5497	0.159022	31.6078
	256^3	256^2	2.95473	0.0514137	0.163731	3.16987
		512^2	2.95442	0.231721	0.164033	3.35017
		1024^2	2.9672	2.28362	0.164222	5.41504
		2048^2	2.95123	14.7621	0.164947	17.8783
Unfavorable Scenario	64^3	256^2	0.331321	1.34332	0.155385	1.83003
		512^2	0.331098	7.32731	0.156015	7.81442
		1024^2	0.329006	38.9989	0.156091	39.484
		2048^2	0.321996	164.36	0.158151	164.84
	128^3	256^2	0.897908	0.671663	0.157064	1.72663
		512^2	0.897689	11.3135	0.157457	12.3687
		1024^2	0.897679	12.5766	0.158837	13.6331
		2048^2	0.896514	49.2111	0.158721	50.2663
	256^3	256^2	2.94273	0.244941	0.16282	3.3505
		512^2	2.97297	2.77733	0.163289	5.91359
		1024^2	2.97265	21.8101	0.163835	24.9466
		2048^2	2.9606	42.8114	0.162547	45.9346

Table 7.5: Performance characteristics when no secondary rays are used.

Light Position	Voxel Grid Resolution	Ray Tracing Resolution	Voxelization Stage (ms)	Radiance Injection Stage (ms)	Final Pass (ms)	Total Rendering Time (ms)
Favorable Scenario	64^3	256^2	0.331848	0.316714	0.15258	0.801141
		512^2	0.33134	2.25671	0.153541	2.74159
		1024^2	0.331435	10.0988	0.154017	10.5843
		2048^2	0.331929	28.9534	0.155079	29.4404
	128^3	256^2	0.89806	0.148795	0.156224	1.20308
		512^2	0.898933	1.00894	0.156432	2.0643
		1024^2	0.898466	7.04459	0.156956	8.10001
		2048^2	0.897514	26.9496	0.157714	28.0049
	256^3	256^2	2.94239	0.0920445	0.161557	3.19599
		512^2	2.94819	0.432613	0.161291	3.54209
		1024^2	2.88405	2.89204	0.161274	5.93736
		2048^2	2.93432	13.5442	0.16196	16.6405
Unfavorable Scenario	64^3	256^2	0.3306	1.49855	0.153396	1.98255
		512^2	0.329876	7.33435	0.154091	7.81832
		1024^2	0.331643	18.3054	0.155164	18.7922
		2048^2	0.330071	104.199	0.155758	104.685
	128^3	256^2	0.898506	0.740024	0.155737	1.79427
		512^2	0.897115	8.29878	0.157151	9.35304
		1024^2	0.89917	12.9854	0.156771	14.0414
		2048^2	0.895979	33.7011	0.156534	34.7536
	256^3	256^2	2.94209	0.269569	0.161115	3.37278
		512^2	2.95074	2.84744	0.161276	5.95945
		1024^2	2.94284	17.1162	0.161626	20.2206
		2048^2	2.96257	30.1941	0.161974	33.3187

Table 7.6: Performance characteristics when a single reflected secondary ray is used.

Sampling Strategy	Videou Output Resolution	Voxel Grid Resolution	Voxelization Stage (ms)	Radiance Injection Stage (ms)	Final Pass (ms)	Total Rendering Time (ms)
Single Sample	640x480	64^3	0.323448	1.99465	0.025762	2.34386
		128^3	0.893852	0.725782	0.0260906	1.64572
		256^3	2.93196	0.206958	0.026776	3.16569
	1280x720	64^3	0.324387	1.95278	0.0372705	2.31444
		128^3	0.894856	0.728571	0.0377954	1.66122
		256^3	2.94085	0.211743	0.0392018	3.19179
	1920x1080	64^3	0.330914	2.0925	0.0576362	2.48105
		128^3	0.898567	0.727715	0.0588111	1.68509
		256^3	2.94662	0.234423	0.0612998	3.24235
7 Samples	640x480	64^3	0.32386	1.99218	0.0372355	0.0372355
		128^3	0.894975	0.727753	0.0379007	1.66063
		256^3	2.93372	0.210851	0.0395772	3.18414
	1280x720	64^3	0.330333	1.96494	0.0799995	2.37528
		128^3	0.89935	0.729266	0.08158	1.7102
		256^3	2.94899	0.214636	0.0849832	3.24861
	1920x1080	64^3	0.331934	1.96789	0.15629	2.45612
		128^3	0.901019	0.717287	0.158172	1.77648
		256^3	2.95833	0.233612	0.163585	3.35553

Table 7.7: Results of testing the effect of the sampling strategy and voxel grid resolution on final pass performance

References

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

[2] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. Image-space horizon-based ambient occlusion. In *SIGGRAPH'08: ACM SIGGRAPH Talks 2008*, 2008.

[3] Chris Brennan. Accurate environment mapped reflections and refractions by adjusting for object distance. *Shader X*, 2002.

[4] Michael Bunnell. Dynamic ambient occlusion and indirect lighting. *GPU Gems*, 2005.

[5] Hao Chen and Xinguo Liu. Lighting and material of Halo 3. In *SIGGRAPH'08: ACM SIGGRAPH Classes 2008*, 2008.

[6] Cyril Crassin. GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes. *Thesis*, 2011.

[7] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum*, 2011.

[8] Carsten Dachsbacher and Marc Stamminger. Reflective shadow maps. In *Proceedings of the Symposium on Interactive 3D Graphics*, 2005.

[9] Scott J Daly. Visible differences predictor: an algorithm for the assessment of image fidelity. In *Human Vision, Visual Processing, and Digital Display III*, volume 1666, pages 2–15. International Society for Optics and Photonics, 1992.

[10] Hawar Doghramachi. Rasterized Voxel-Based Dynamic Global Illumination. In *GPU Pro 4*. 2013.

[11] Ahmet M. Eskicioglu and Paul S. Fisher. Image Quality Measures and Their Performance. *IEEE Transactions on Communications*, 1995.

[12] Alex Evans. Fast approximations for global illumination on dynamic scenes. In *SIGGRAPH 2006 - ACM SIGGRAPH 2006 Courses*, 2006.

[13] Bernd Girod. *What's Wrong with Mean-Squared Error?*, page 207–220. MIT Press, Cambridge, MA, USA, 1993.

[14] Otavio Good and Zachary Taylor. Optimized photon tracing using spherical harmonic light maps. In *ACM SIGGRAPH 2005 Sketches, SIGGRAPH 2005*, 2005.

[15] Ralf Habel and Michael Wimmer. Efficient irradiance normal mapping. In *Proceedings of I3D 2010: The 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2010.

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

[17] Antti Hirvonen, Atte Seppälä, Maksim Aizenshtein, and Niklas Smal. Accurate real-time specular reflections with radiance caching. In *Ray Tracing Gems*, pages 571–607. Springer, 2019.

[18] Jared Hoberock and Yuntao Jia. High-quality ambient occlusion. *GPU gems*, 3:257–274, 2007.

[19] ITU. Methodology for the subjective assessment of the quality of television pictures, 2002.

[20] Michal Iwanicki. Lighting technology of the last of us. 2013.

[21] Ruud Janssen. *Computational image quality*, volume 101. SPIE press, 2001.

[22] Xianchun Wu Angelo Pesce Jimenez, Jorge and Adrian Jarabo. Practical Realtime Strategies for Accurate Indirect Occlusion. In *Siggraph 2016*, 2016.

[23] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1986*, 1986.

[24] Anton Kaplanyan. Light Propagation Volumes in CryEngine 3. *ACM SIGGRAPH Courses*, 2009.

[25] Anton Kaplanyan and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of I3D 2010: The 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2010.

[26] L. Kavan, A. W. Bargteil, and P. P. Sloan. Least squares vertex baking. *Computer Graphics Forum*, 2011.

[27] Alexander Keller. Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1997*, 1997.

[28] O. Klehm, T. Ritschel, E. Eisemann, and H. P. Seidel. Bent normals and cones in screen-space. In *VMV 2011 - Vision, Modeling and Visualization*, 2011.

[29] Janne Kontkanen and Samuli Laine. Ambient occlusion fields. In *Proceedings of the Symposium on Interactive 3D Graphics*, 2005.

[30] Sébastien Lagarde and Charles de Rousiers. Moving Frostbite to PBR. *SIGGRAPH 2014*, 2014.

[31] Hayden Landis. Production-ready global illumination. *Siggraph course notes*, 2002.

[32] Jaakko Lehtinen, Matthias Zwicker, Emmanuel Turquin, Janne Kontkanen, Frédo Durand, François X. Sillion, and Timo Aila. A meshless hierarchical representation for light transport. *ACM Transactions on Graphics*, 2008.

[33] Eric Lengyel. Volume 2: Rendering. In *Foundations of Game Engine Development*, pages 4–21. Terathon Software, 2019.

[34] Bradford James Loos and Peter Pike Sloan. Volumetric obscurance. In *Proceedings of I3D 2010: The 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2010.

[35] Jeffrey Lubin. The use of psychophysical data and models in the analysis of display system performance. In *Digital images and human vision*, pages 163–178. MIT Press, 1993.

[36] Jeffrey Lubin. A visual discrimination model for imaging system design and evaluation. In *Vision Models for Target Detection and Recognition: In Memory of Arthur Menendez*, pages 245–283. World Scientific, 1995.

[37] Zander Majercik, Jean-Philippe Guertin, and Morgan Mcguire. Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields. *Journal of Computer Graphics Techniques Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields*, 2019.

[38] Mattias Malmer, Fredrik Malmer, Ulf Assarsson, and Nicolas Holzschuch. Fast Precomputed Ambient Occlusion for Proximity Shadows. *Journal of Graphics Tools*, 2007.

[39] James L. Mannos and David J. Sakrison. The Effects of a Visual Fidelity Criterion on the Encoding of Images. *IEEE Transactions on Information Theory*, 1974.

[40] Nelson L. Max. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer*, 1988.

[41] Morgan McGuire, Mike Mara, Derek Nowrouzezahrai, and David Luebke. Real-time global illumination using precomputed light field probes. In *Proceedings - I3D 2017: 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2017.

[42] James McLaren. The technology of the tomorrow children.

[43] Kenny Mitchell, Peter Pike Sloan, Bradford J. Loos, Lakulish Antani, Derek Nowrouzezahrai, and Wojciech Jarosz. Modular Radiance Transfer. *ACM Transactions on Graphics*, 2011.

[44] Martin Mittring. Finding next gen - CryEngine 2. In *ACM SIGGRAPH 2007 Papers - International Conference on Computer Graphics and Interactive Techniques*, 2007.

[45] David Neubelt and Matt Pettineo. Advanced lighting r & d at ready at dawn studios. Technical report, SIGGRAPH Advances in Real-Time Rendering in Games course, 2015.

[46] Chris Oat and Pedro Sander. Ambient aperture lighting. In *SIGGRAPH 2006 - ACM SIGGRAPH 2006 Courses*, 2006.

[47] Stephen E Palmer. *Vision science: Photons to phenomenology*. MIT press, 1999.

[48] Alexey Panteleev. Practical real-time voxel-based global illumination for current gpus. In *ACM SIGGRAPH 2014 presentations*, 2014.

[49] Zhong Ren, Rui Wang, John Snyder, Kun Zhou, Xinguo Liu, Bo Sun, Peter Pike Sloan, Hujun Bao, Qunsheng Peng, and Baining Guo. Real-time soft shadows in dynamic scenes using spherical harmonic exponentiation. In *ACM SIGGRAPH 2006 Papers, SIGGRAPH '06*, 2006.

[50] Michael Schwarz. Practical binary surface and solid voxelization with direct3D 11. In *GPU PRO3: Advanced Rendering Techniques*. 2012.

[51] Peter Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '02*, 2002.

[52] Tiago Sousa. Adaptive glare. *Shader X3: Advanced Rendering with DirectX and OpenGL*, pages 349–355, 2005.

[53] Eric Tabellion and Arnauld Lamorlette. An approximate global illumination system for computer generated films. In *ACM SIGGRAPH 2004 Papers, SIGGRAPH 2004*, 2004.

[54] Masaya Takeshige. The basics of gpu voxelization, Mar 2015.

[55] P. C. Teo and D. J. Heeger. Perceptual image distortion. In *Proceedings - International Conference on Image Processing, ICIP*, 1994.

[56] Ville Timonen. Line-sweep ambient obscurance. *Computer Graphics Forum*, 2013.

[57] Jiaping Wang, Peiran Ren, Minmin Gong, Baining Guo, Peiran Ren, Baining Guo, and John Snyder. All-Frequency Rendering of Dynamic, Spatially-Varying Reflectance. *ACM Transactions on Graphics*, 2009.

[58] Zhou Wang and Alan C. Bovik. A universal image quality index. *IEEE Signal Processing Letters*, 2002.

[59] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[60] Andrew B Watson. Dct quantization matrices visually optimized for individual images. In *Human vision, visual processing, and digital display IV*, volume 1913, pages 202–216. International Society for Optics and Photonics, 1993.

[61] Daniel Wright. Dynamic occlusion with signed distance fields. Technical report, SIGGRAPH Advances in Real-Time Rendering in Games course, 2015.

[62] Kun Xu, Wei Lun Sun, Zhao Dong, Dan Yong Zhao, Run Dong Wu, and Shi Min Hu. Anisotropic spherical gaussians. *ACM Transactions on Graphics*, 2013.

[63] S. Zhukov, A. Iones, and G. Kronin. An ambient light illumination model. 1998.