Real-time Global Illumination using Direct3D

Bjarke Damsgaard Eriksen, 201708652

Aarhus University 2021
Supervisor: Assistant Professor Jung Min Kim
Total character count: 75602

Aarhus University
Department of Electrical and Computer Engineering
Katrinebjerg
Finlandsgade 22 8200 Aarhus N, Denmark
Phone +45 8715 0000
ece@au.dk
www.ece.au.dk

Abstract (English)

Global Illumination (GI) is an influential component of rendering believable imagery. Traditional methods like path tracing, which achieves GI by shooting millions of rays into a 3D scene, are unsuitable as their computational overhead exceeds real-time budgets. In this bachelor's thesis, we analyze and implement a real-time method called voxel cone tracing that approximates global illumination by tracing a series of increasingly larger spheres into a 3D scene. Furthermore, we investigate and implement additional rendering techniques to enhance the quality of the rendered scene with shadows and normal mapping. We specify functional and non-functional requirements for the application in relation to the problem's domain, which we use to assess the quality of the implementation where we pass 21/26 tests.

In the analysis of our voxel cone tracing solution, we conclude that our implementation successfully approximates global illumination in real-time. However, we find that the performance and visual quality could be improved in a plethora of ways, which is consistent with our understanding of the current literature.

Abstract (Danish)

Global illuminering (GI) er en af de mest altafgørende komponenter, når man skal gengive troværdige billeder. Traditionelle metoder, såsom path tracing, opnår GI ved at skyde flere millioner af rays ind i en 3D scene, og dette er uholdbart for reel-tidsapplikationer, da det beregningsmæssigt er for dyrt. I denne bacheloratafhandling analyserer og implementerer vi en reel-tidsmetode kaldet voxel cone tracing, som approksimerer global illumination ved at trace en serie af sfærer i stigenende størrelse ind i en 3D-scene. Ydermere undersøger og implementerer vi supplerende gengivelsesmetoder for at forbedre billedekvaliteten af den gengivne scene med skygger og normal mapping. Vi specificerer funktionelle og ikke-funktionelle krav for applikationen i relation til problemformuleringens domæne, som bruges til at vurdere kvaliteten af implementeringen, hvor vi består 21/26 tests.

I analysen af vores voxel cone tracing-løsning konkluderer vi, at vores implementering med succes approksimerer global illuminering i reeltid. Vi oplever dog, at ydeevnen og billedekvaliteten kan blive forbedret på adskillige måder, hvilket er i overensstemmelse med vores forståelse af den aktuelle litteratur.

Acknowledgements

I would like to thank my supervisor, Assistant Professor Jung Min Kim, for allowing me to explore a topic that I find immensely interesting and for going above and beyond in helping me with the thesis and my studies. Furthermore, I wish to express my sincere gratitude to my dear friend Gian Schellenbaum for being an invaluable discussion partner, encouraging me to grow and enabling me to learn such an enormous topic, and helping me improve the quality of the thesis. I would also like to thank Jacob Falgren Christensen, B.Sc in Engineering (Robot Systems), for reviewing and giving feedback on the report.

Contents

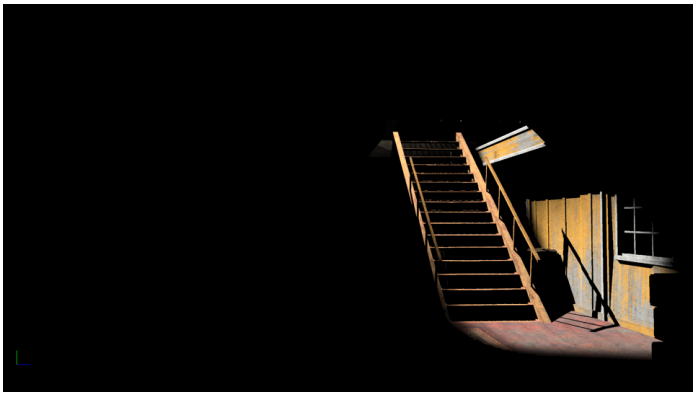
Chapter 1

Introduction

1.1 Global Illumination

In this bachelor thesis, several methods for achieving global illumination in real-time will be presented. These methods include voxel-based ambient occlusion and indirect lighting. Supplemental techniques are added for more realistic lighting and shadows. These techniques include shadow mapping, direct lighting, direct and indirect specular, and more. These algorithms exist in many video games, computer-aided design (CAD), virtual reality (VR), and augmented reality (AR) applications, where they are essential for increasing the realism or believability of Computer Generated Imagery (CGI).

Believable lighting is the holy grail of computer graphics. Light doesn't only come directly from a light source. In the real world, light can get reflected off of surfaces and illuminate surfaces that aren't directly visible from a light source. This light contribution is called indirect lighting. Without it, there wouldn't be any light where a light source isn't directly visible, as illustrated by Figure 1.1. This is where effects such as color bleeding come from. Global illumination is the total contribution of direct light and indirect light.



(a) Direct lighting only.

(b) Direct and indirect lighting combined.

Figure 1.1: Images courtesy of Silvennoinen et al. [ST15].

1.2 Problem Statement

Global illumination has existed for a while in many different applications like video games, movies, and architectural development tools. Today CGI is used in most film productions, where actors perform in film studios covered in blue sheets wearing motion tracking suits in order to get their performance digitally transcribed in post-production. These days, rendering a single image of a blockbuster may take several hours or days on dedicated rendering systems [Tak13]. It isn't odd then that developers have been seeking to reduce the computational overhead of the rendering process. In real-time applications, this is especially true as they likewise want to offer a grounded and realistic experience, with a frame-time budget of 33 milliseconds for 30 frames per second (FPS) or 16 milliseconds for 60 FPS.

To solve this, algorithms that achieve photorealism by approximation have been developed and are an active area of research. Some of these techniques, such as the one presented in this thesis, approximate indirect lighting by tracing cones, successfully lowering computational overhead at the cost of precision. Our thesis concentrates on a method known as voxel cone tracing, first described by Crassin et al. [Cra+11]. It gets used in conjunction with other algorithms to render a believable real-time scene. We will compare and analyze different ways to implement voxel cone tracing and assess its performance, memory usage, and image quality. We seek to provide a detailed description of the used methods and their implementation. Note that our approach is not a final solution for all of global illumination.

1.3 Process Method

The thesis was developed in an iterative and exploratory manner. Agile development techniques such as scrum were discarded since they didn't make sense. For source versioning, we used Git with a mostly trunk-based development process, occasionally using branches for more experimental endeavors. A lot of tools have been used during development, such as Visual Studio for code editing and debugging, Git for version control, clang-format for formatting the code, GENie for writing build scripts, RenderDoc for graphics debugging, NVIDIA Texture Tools Exporter for texture compression and mip map generation, and Blender for editing and exporting the geometry and materials.

1.4 Outline

Chapter 2: Requirements
Defines the functional and non-functional requirements in relation to the problem statement of the bachelor thesis. Declares the necessary acceptance tests that need to pass for the thesis to be deemed successful.

Chapter 3: Technical Analysis
Describes the development process, the tools involved, and the reasoning behind their use. Moreover, programming techniques and algorithmic choices are compared and briefly analyzed.

Chapter 4: Architecture
Describes the architecture of the application that was developed to achieve the goals of the thesis. Design choices are explained and analyzed for the domain of the problem statement.

Chapter 5: Design and Implementation
Explains the design and implementation of the different modules in detail, with illustrations, code snippets, and formulas. Examines rendering techniques and analyses their implementation in relation to performance, image quality, and memory usage.

Chapter 6: Acceptance Test
Presents the outcome of the application's acceptance test of its functional and non-functional requirements and provides a brief discussion of the overall results.

Chapter 7: Results and Discussion
Shows the system setup used to render the scene when capturing measurements. Highlights and discusses important performance impacting settings and optimization used when rendering. Provides an analysis of various voxel cone tracing related performance metrics and the measurements.

Chapter 8: Conclusion
Summarizes the results and compares our voxel cone tracing solution to the one put forth by Crassin et al. [Cra+11]. Future work ideas for enhancements, optimizations, and experimentation are detailed.

Chapter 2

Requirements

Before we take a look at the Architecture and Design of the rendering engine, we need to list the requirements of the bachelor thesis. The requirements provide the basis for how to program needs to be structured. Thus, they need to be formally defined. These requirements specify the goals that need to be realized for the project to be successful. The system is formally defined as the renderer, rendering engine, or simply the application.

2.1 Use Cases

There are two primary use case scenarios for the application: the first is startup initiated by a user, after which the test scene starts rendering. Then, the second scenario describes how the renderer updates based on user input.

Name	UC1: Application startup
Goal	The renderer is initialized on the user's PC.
Actors	Prime: User.
Initialization	User opens the application.
Precondition	The user has installed the renderer as per instructions.
Postcondition	The renderer displays the default 3D model with Global Illumination and a GUI.
Main scenario	1. The user opens the renderer on their computer. 2. The renderer shows the default 3D-model.
Extensions	N/A
Special Requirements	Adequate hardware – must meet system requirements.
Miscellaneous	N/A

Table 2.2: UC1: Application startup

Name	UC2: Scene manipulation
Goal	The scene is transformed based on user input.
Actors	Prime: User.
Initialization	User uses the GUI or navigation keys.
Precondition	The renderer is running.
Postcondition	The scene has been transformed and Global Illumination is updated accordingly.
Main scenario	1. User uses the GUI to change scene parameters, or moves the camera around the scene. 2. The renderer reads the input and uses it to change the scene. 3. The user gets visual feedback from the renderer.
Extensions	N/A
Special Requirements	Keyboard and mouse – so the user can move the camera.
Miscellaneous	N/A

Table 2.4: UC2: Scene manipulation

2.2 Functional requirements

We need to specify the functional requirements to designate what the application should do and which features to implement. We do this to identify potential issues and to save time and resources. This way, it's possible to give a rough estimate of development time and potential bottlenecks that may require additional thought.

Test Definition
The renderer must implement ambient occlusion.
The renderer must implement shadow mapping.
The renderer must implement indirect lighting.
The renderer must implement normal mapping.
The renderer must not have any dependencies that aren't part of a base Windows installation (Direct3D 11 excluded).
The renderer should have data pre-processing for optimizing assets.
The renderer could support contact-hardening soft shadows.
The renderer could support multiple bounces of indirect lighting.
The renderer could support multiple emittance computation and storage modes.

Table 2.6: Functional requirements of the rendering engine.

2.3 Non-functional requirements

We list a set of non-functional requirements that specify how to assess the operation of the application after delivery. These conditions impose constraints on the implementation of requirements for fundamental characteristics such as performance, reliability, and scalability.

Group	NFR No.	Non-functional requirement
Usability		
	U1	The application will allow changing all important settings at run-time through a GUI.
	U2	The application will use Dear ImGui to draw the graphical user interface.
	U3	It takes a maximum of five minutes to understand the GUI .
	U4	The user will be able to interact with the 3D scene and move around in the world.
	U5	The application will have debug visualization modes.
	U6	The application will have built-in facilities for assessing performance and memory usage.
	U7	The application will allow changing of all light properties at any time.
Reliability		
	R1	The application has an up-time (MTBF) of at least 99 percent.
	R2	The application has a mean-time-to-repair of 3 hours.

Group	NFR No.	Non-functional requirement
Performance		
	P1	The application will have an average frame time of 60 frame-per-second on the minimum required hardware.
	P2	It takes less than two seconds to load the scene.
	P3	The response time of the application when receiving input must seem instantaneous.
Supportability		
	S1	The application is compatible with Windows 7/8/10.
	S2	The application runs on all devices that support Direct3D feature level 11.0.
	S3	The application is expandable with new features on the basis that the developer understands the architecture.
	S4	The GUI is expandable with additional content e.g. to improve user experience.
	S5	The renderer must run on hardware that doesn't support conservative rasterization.

Table 2.9: Non-functional requirements of the rendering engine. Listed in groups. Usability is the degree of ease with which the user interacts with the renderer to achieve their goal. Reliability defines the metrics that show the possibility of the rendering engine failing. Performance declares how the rendering engine performs when users interact with it in various scenarios. Supportability shows the configurability and extendability of the rendering engine.

2.4 Acceptance test specification

The requirements need to be testable and verifiable, if they aren't then we cannot determine if the bachelor thesis is successfully solving the problem statement. In the following two sections, we define step-for-step how to test each functional and non-functional requirement.

2.4.1 Functional requirements

No.	Action	Expected Result	Result	Passed
1	Enable 'Only Ambient Occlusion' from the GUI.	Scene is only drawn with AO.	N/A	N/A
2	Enable either PCSS or PCF filtering and go to an occluded surface.	Shadows are drawn in the occluded area.	N/A	N/A
3	Enable 'Indirect Diffuse' and 'Indirect Specular'.	Scene is illuminated by indirect light.	N/A	N/A
4	Enable 'Use normal map.'	Bumps and dents in surfaces are added to the scene.	N/A	N/A
5	The developer will show that there are no dependencies.	N/A	N/A	N/A
6	The developer will show that the data was preprocessed.	N/A	N/A	N/A
7	Enable 'PCSS' shadows and inspect the shadows.	Shadows are contact-hardening.	N/A	N/A
8	Enable 'Indirect Diffuse' and 'Indirect Specular'. Change the slider that dictates the number of light bounces.	If the slider is increased, the scene becomes brighter. If the slider is decreased, the scene becomes darker.	N/A	N/A
9	Switch between HDR modes.	Performance and quality change depending on the HDR mode.	N/A	N/A

Table 2.11: Acceptance test specification for functional requirements.

2.4.2 Non-functional requirements

No.	NFR	Action	Expected Result	Result	Passed
1	U1	Assuming that the GUI is enabled. Change settings.	The drawing of the scene changes in correspondence to the changed setting.	N/A	N/A
2	U2	Assuming that the GUI is enabled, see that it is drawn with Dear ImGui.	The GUI is drawn using Dear ImGui.	N/A	N/A
3	U3	Inspect the GUI.	It takes less than five minutes to understand the GUI.	N/A	N/A
4	U4	Use WASD and the mouse to move around the scene.	The camera moves around in response to user input.	N/A	N/A
5	U5	Enable debug visualization modes through the GUI.	Debug visualization modes are drawn.	N/A	N/A
6	U6	Inspect the GUI and locate performance and memory usage statistics.	It's possible to assess performance and memory usage.	N/A	N/A
7	U7	Inspect the GUI and locate light settings. Change the settings.	Lights are updated and the look of the scene changes.	N/A	N/A
8	R1	N/A	N/A	N/A	N/A
9	R2	N/A	N/A	N/A	N/A

No.	NFR	Action	Expected Result	Result	Passed
10	P1	Inspect the GUI and locate the frame-time.	FPS should be at an average of 60 or higher.	N/A	N/A
11	P2	Measure the time it takes to start the application.	It should take less than two seconds to load the scene.	N/A	N/A
12	P3	Use WASD and the mouse to move around the scene.	The response seems to input feels instantaneous.	N/A	N/A
13	S1	Load the Application on Windows 7/8/10	The application loads.	N/A	N/A
14	S2	Run the application on multiple devices that support Direct3D feature level 11.0	The application starts and runs fine.	N/A	N/A
15	S3	The developer explains the architecture and expandability of the application.	The reasoning given by the developer is reliable.	N/A	N/A
16	S4	The developer explains how the GUI works and that it is easily expandable.	The reasoning given by the developer is reliable.	N/A	N/A
17	S5	Run the application on hardware that doesn't support conservative rasterization. Enable software conservative rasterization.	A software implementation of conservative rasterization is used during voxelization.	N/A	N/A

Table 2.14: Acceptance test specification for non-functional requirements.

Chapter 3

Technical Analysis

3.1 Code Conventions

Coding conventions serve as guidelines for the structural quality of software. They make it easier to focus on the content and enable quick understanding, and demonstrate best practices. Furthermore, they facilitate reading, sharing, and maintaining code. These conventions typically cover file organization, programming practices and principles, and formatting rules.

In this thesis, coding conventions are important since the development language is C++, which has become a very complex standard. Real-time applications cannot be built on top of a slow and hard-to-maintain code base. Therefore, we only use a subset of C++'s features to keep things simple and decrease cognitive load.

We do not use the C++ Standard Library. We rely on our implementation of data structures and resort to the C standard library for some functionality. Some of the C++ features that we use are operator overloading, templates for data structures and small functions, default argument values, static assertions, constructers and destructors. Furthermore, we do not use Object Oriented Programming (OOP).

3.2 Direct3D 11

There exist a number of APIs for applications that need to render 2D and 3D images. For Windows, there's Direct3D, OpenGL, and Vulkan. Direct3D is API of choice on Windows and Xbox and is controlled by Microsoft, while OpenGL and Vulkan are cross-platform open standards handled by the Khronos Group. Direct3D and OpenGL have been competing on Windows for a long time, and Vulkan has seen a surge in users over the last couple of years.

Vulkan is the API of choice on Android. It is cross-platform on the desktop. However, as with some of the other APIs, Vulkan doesn't run on any consoles. It is known to be hard to set up, especially for people new to computer graphics development. OpenGL is the most commonly used graphics API on Unix-like systems. Furthermore, neither iOS nor MacOS are supported for either API.

Direct3D runs on Windows Desktop and Xbox and has some embedded system support through Direct3D Mobile. It's well documented[1], drivers are validated, and tool support is generally excellent, which makes it easy for new users to get started with graphics programming.

3.3 Operating System

The choice of the operating system is a no-brainer. Since we have chosen to work with Direct3D, we're forced to use Windows. Fortunately, Windows has the best desktop development environment for graphics applications, thanks to the available libraries, tooling and documentation.

We have chosen to work with the Win32 API, which provides direct access to Windows and the hardware. According to Microsoft, it's the platform of choice for applications that need the highest level of performance and direct access to system hardware. Many external libraries, like the one addressed in the next section, provide documentation and examples using Win32 which makes it easier to get started.

3.4 Dear ImGui

Rendering graphical user interfaces (GUIs) for applications is a big topic, which itself could be a bachelor thesis. It isn't valuable to implement a GUI API as part of our system because we do not learn anything about the topic at hand and it is simply time consuming. Dear ImGui [Cor] is a bloat-free C++ GUI library with minimal dependencies. It's an immediate mode GUI, which means that no UI element needs to be declared nor registered before use. It issues rendering commands each frame when required but it batches draw calls very efficiently. The resulting number of draw calls is roughly one per UI depth layer. In a retained mode UI, the library operates on its own copy of the data or through some data binding mechanism. Dear ImGui just uses the data directly from function calls. The library enables fast iteration times and is easy to integrate,

[1] https://microsoft.github.io/DirectX-Specs/, The full Direct3D 11 & 12 specifications.

hence our choice to use it.

3.5 RenderDoc

RenderDoc [Kar18] is a graphics debugging tool that allows us to capture and replay a single frame of an application using Direct3D 11 and other supported APIs. With this, it is possible to analyze what happens during the specific captured frame. The state of textures, buffers, pipeline stages, etc. can be inspected at any point during the frame. It is also possible to step through shader code instruction by instruction. Just like we use our C++ debugger to step through the application's code and inspect memory, we can use RenderDoc to do the same on the GPU. Our application uses the Direct3D API to annotate resources and events so that RenderDoc can display more easily understandable names for resources and groups of draw calls.

3.6 Build System

GENie [Kar+] is a command-line Makefile and project file generation tool. For our application its input is a single Lua script file describing the build and its output is a Visual C++ project. Instead of having to manually maintain several Visual C++ projects by hand, we edit the Lua script once and regenerate all Visual C++ projects from it.

3.7 Texture File Encoding and Compression

Texture encoding and compression are crucial in real-time applications. Texture compression allows us to reduce the size of our textures thereby decreasing load times and fitting more data in memory. Being able to fit additional data in memory could make it possible to render larger worlds or larger sections of the world and streaming textures in faster or with greater detail since compressed textures will be transferred faster from main memory to GPU memory. Another benefit of compressed texture is the reduced memory bandwidth usage, which translates to increased performance.

NVIDIA Texture Tools Exporter (NVTTE) [NVI] is a tool that can compress textures with various attractive features. In our system, all textures have been pre-processed using NVTTE with different mip filters and compression settings based on intended usage (albedo, specular, normal).

It our application, we only use BCn texture formats. BCn is a family of lossly GPU-supported block-based texture compression formats.

For albedo textures, we use BC7 which uses eight bits per pixel, i.e. a 4:1 compression ratio. We encode specular textures as BC4u, which is a grayscale format where only the red channel is kept. It uses four bits per pixel, and has a 2:1 compression ratio. The application still supports RGB reflectance data at the material level instead of the per-pixel level.

We use BC5u for normals, which also uses eight bits per pixel and only uses two channels. This is fine because we do not have to store the z component of the normal. This comes from the fact that the z value can be calculated as:

$$\begin{aligned} x^2 + y^2 + z^2 &= 1 \\ z^2 &= 1 - x^2 - y^2 \\ z &= \sqrt{1 - x^2 - y^2} \end{aligned} \tag{3.1}$$

The loss of the sign from the square root is not an issue since we use tangent-space normal maps: the normal always points up and therefore the z value is always positive. Otherwise, we would have to use one more bit to know the sign of the z value.

Now we know how to compress the data but we still need to choose how to store it on disk. We picked the DirectDraw Surface (DDS) format because it has built-in support for all the format enumerations from Direct3D, and storage of the entire mip chain in a single file. Another option could have been KTX, as it provides the same features and additional lossless compression on top as well.

3.8 Forward and Deferred Rendering

In a forward renderer, all shading is done while drawing the geometry. One of the major disadvantages of forward rendering is overdraw, which is the amount of fragments wasted computing the final value for a surface that later becomes hidden by another pixel shader implication. Overdraw happens when triangles are rendered on top of each other and the final draw order isn't back-to-front or when triangles intersect. For example, if the camera is facing a solid wall with a lot of geometry behind it, you could render the whole scene behind the wall first, but none of it is visible in the end. Since lighting computations and many texture fetches happen in the pixel shader, the potential performance impact of overdraw can become very high.

In deferred rendering [Dee+88; Cal03], geometry rendering and shading for opaque surfaces are decoupled. First, the geometry is rendered and attributes are written to a set of textures commonly known as the G-buffer (as in geometry

buffers). Then, the deferred pass runs a full-screen pixel or compute shader invocation that reads from the G-buffer, shades, and writes the result to a render target. A G-buffer typically contains attributes such as albedo color, surface normals, and specular reflectance. Thanks to the delayed (deferred) shading step, only visible pixels are shaded, which means that there is no overdraw. A major drawback of this approach is that it doesn't work with multisample anti-aliasing (MSAA) because the resolve step needs to happen after shading but the resolve of certain buffers such as the surface normals would be nonsensical. Even if solved, the increased memory and bandwidth requirements due to the use of multi-sampled G-buffer textures would be prohibitive. Instead, modern games that use variants of deferred rendering make use of post-processing anti-aliasing techniques, chiefly temporal anti-aliasing (TAA) [Yan+09; Sou11; Kar14].

It should be noted that there are many more variants used in real-time rendering, such as light pre-pass [Eng09], Forward+ [HMY12], visibility buffer rendering [BH13], and object-space shading [CCC87; Bak16].

Chapter 4

Architecture

The application consists of three modules. A module consists of code responsible for handling specific tasks. Each module has a private and public interface. The public interface provides functions, data structures, and more to other modules which might need them, while the private interface does the same but is only accessible to the module itself. Furthermore, each module is part of a layer. Our application consists of two layers: the application layer is for modules that have no dependency on the operating system, while the system layer relies on the OS for functionality.

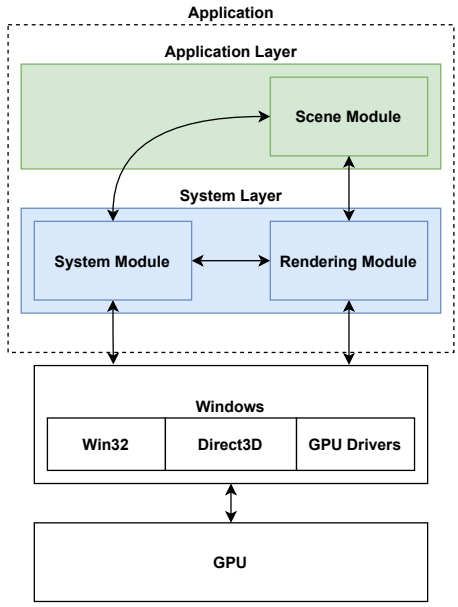
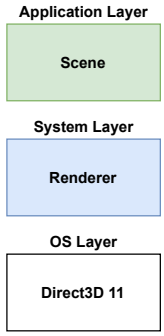
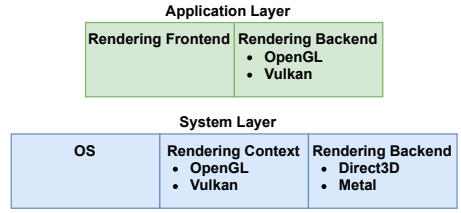


Figure 4.1: The logical structure of the system.

We have a simple architecture since we only ship on one operating system. We could have supported multiple rendering APIs by having a stronger division between renderer frontend and backend where only the backend interacts with a graphics API. In this scheme, the frontend generates a queue of work items for the backend to consume, see Figure 4.2.



(a) Our architecture, which has strong coupling between rendering and the operating system services.



(b) Division between renderer frontend and backend that allows for a cross-platform renderer.

Figure 4.2: The rendering context is the glue between the graphics API and the application window. The Vulkan API calls it a surface. While the Vulkan and OpenGL APIs are mostly platform independent, a small amount of platform specific code is needed to allow the APIs to draw to a window. In addition, platform specific extensions might be needed to handle other concerns such as vertical synchronization.

4.1 System Module

The system module is part of the system layer as it abstracts the operating system. It provides access to file reads, timings, and more system services through its public interface. It is responsible for setting up the application window and initializing the rest of the application to start rendering. Furthermore, it handles input from the user and forwards it to the scene module. It is important to note that the system module doesn't issue any rendering commands. Its only purpose is to expose the aforementioned utilities, forward user input, and provide memory to the other modules.

There's a couple of ways to implement a system module, each with its pros and cons. One way is listing everything the application needs, abstracting it and providing an interface for it. The other is to only abstract the specific platform utilities needed in order to get going and not provide an API for the rest. As an example, we will construct a window to draw into using the two schemes. In the case of abstracting a window, we would have to provide an API for creating and destroying a window at a minimum. Depending on the application's needs, the API could extend further to deal with resizing, initializing the rendering context, toggling full-screen mode, etc. All these abstractions quickly add up, and soon we would have created a large abstraction that's difficult to maintain and where we only require a couple of features. On the other hand, we could abstract nothing and set up the window as we want it in the system module itself. If we need to change something about the window, we will keep that logic in the module since it already is only handling Windows-specific code. This way, we keep everything simple and hold platform-specific code away from other modules as much as possible by keeping the system interface as small as it can be.

For our application, we need to create abstractions for files to allow modules to read and write to disk, e.g. the scene module reading asset files or the rendering module reading textures. Then, we also expose generalized timing functions, which we use for debugging and optimization during development.

4.2 Scene Module

The scene module acts as the scene manipulator and is part of the application layer as it has no dependency on the operating system. It is responsible for loading assets from disk into memory and then queuing them for rendering. Furthermore, it's responsible for adding lights to the 3D scene and specifying the matrices needed for the lights and the camera used when rendering. It receives input from the system module, which controls the camera in the world.

4.3 Rendering Module

The rendering module is the most prominent. It consist of two minor modules, one is for general purpose rendering and the other handles everything related to voxels. It's part of the system layer since it requires Direct3D to access the GPU. The main responsibility of the rendering module is drawing the data specified by the scene module. It does this by setting up the rendering and compute pipelines and then dispatching draw and compute calls to the GPU. The final result is then displayed in the application's window.

The rendering module has a cache, which is there to reduce API call latency when switching state in the rendering or compute pipeline. This state switching is a product of render passes. In our renderer we need to use multi-pass rendering techniques, but since some of these share state, we do not want to produce latency by binding an already set state.

Chapter 5

Design & Implementation

In this chapter, we discuss the details of our design and implementation. We will be going in-depth about all the shortcomings of each algorithm, presenting edge and failure cases. We will showcase how the GPU receives work from the CPU, detailing the data flow. We will use formulas, pseudo-code and shader examples to illustrate how voxelization, data processing and voxel cone tracing work. In addition, we will also show the implementation of ambient occlusion, indirect lighting and specular lighting. Finally, the chapter also examines the implementation of direct lighting, normal mapping and shadow mapping.

5.1 Physical Structure

To create a clear delimitation between modules and their functionality, source files and headers are subdivided into separate folders identified by their module name e.g. source files related to the renderer, are placed in a 'renderer' directory. By doing this, we reinforce the logical structure of the system.

5.2 Scene

In the scene module, we load assets, handle input forwarded from the system module, and queue work for the rendering module. This section provides a brief overview of how these three different processes are implemented.

5.2.1 Asset Loading

The application loads assets in our custom binary scene format. We have pre-processed the Sponza scene by McGuire [McG16], with our own code to optimize the data layout and compute new smoothed vertex normals. The scene converter reads in a Wavefront OBJ geometry file and its corresponding material definition file and outputs a single file in our optimized format.

As part of the pre-processing of data, we do normal smoothing. Assuming that there are no duplicated vertex positions in the data set, it's possible to run a $O(n)$ smoothing algorithm. If this isn't the case, the final result will be wrong and we have to resort to a alternative approach which runs in $O(n^2)$.

```
# This is the simplest approach but vertices can't have duplicate positions.
↪  Otherwise, we'd fail to add to some vertices and the final normalized
↪  result can have the wrong direction. This is perfectly viable when you
↪  have control over your data and can guarantee no position duplicates.
def smooth_normals_fast(mesh):
    for sg in mesh.smoothingGroups:
        for v in sg.vertices:
            v.normal = float3(0, 0, 0)
        for t in sg.triangles:
            p0 = t.vertices[0].position
            p1 = t.vertices[1].position
            p2 = t.vertices[2].position
            N = cross(p2 - p0, p1 - p0)
            t.vertices[0].normal += N * angle(p1 - p0, p2 - p0)
            t.vertices[1].normal += N * angle(p2 - p1, p0 - p1)
            t.vertices[2].normal += N * angle(p0 - p2, p1 - p2)
        for v in sg.vertices:
            v.normal = normalize(v.normal)
```

Listing 1: A $O(n)$ vertex normal smoothing algorithm that assumes no position duplicates.

```
# The following approach works in all cases.
def smooth_normals_robust(mesh):
    for sg in mesh.smoothingGroups:
        for v in sg.vertices:
            v.normal = float3(0, 0, 0)
            for t in sg.triangles:
                p0 = t.vertices[0].position
                p1 = t.vertices[1].position
                p2 = t.vertices[2].position
                N = cross(p2 - p0, p1 - p0)
                if p0 == v.position:
                    v.normal += N * angle(p1 - p0, p2 - p0)
                elif p1 == v.position:
                    v.normal += N * angle(p2 - p1, p0 - p1)
                elif p2 == v.position:
                    v.normal += N * angle(p0 - p2, p1 - p2)
            v.normal = normalize(v.normal)
```

Listing 2: A $O(n^2)$ vertex normal smoothing algorithm that makes no assumption.

As an optimization, we also have a lower level of detail version of the Sponza scene that is used exclusively for the voxelization process. It contains roughly four times fewer triangles, and the smallest ones are replaced by bigger ones. This matters most because a single voxel can contain a lot of small triangles, which would generate a lot of fragments to process in our atomic operations. We will detail our usage of atomic operations in Section 5.3.6.4.

5.2.2 Input Handling

The system module captures the state of the mouse and keyboard and forwards it to the scene module. In our implementation, we always keep the state of the previous and current frame's input. Then, we use the data to toggle functionality on down transitions, rotate the camera in response to mouse movement, and move the camera when a key is down.

5.2.3 Rendering Command Queue

The scene module sets up work for the renderer by writing into a first in, first out (FIFO) command queue. The queue is a contiguous memory region backed by a simple linear allocator. Each rendering command is defined by its own data structure where the first four bytes encode the command type. These types include scenes, boxes for debugging visualization, and debug regions used for debugging with RenderDoc. A visualization of the data layout can be seen in Figure 5.1. At the start of a frame, the scene module clears the queue and pushes its work items. The renderer then processes every entry in order using

the graphics API to set state and dispatch work to the GPU.

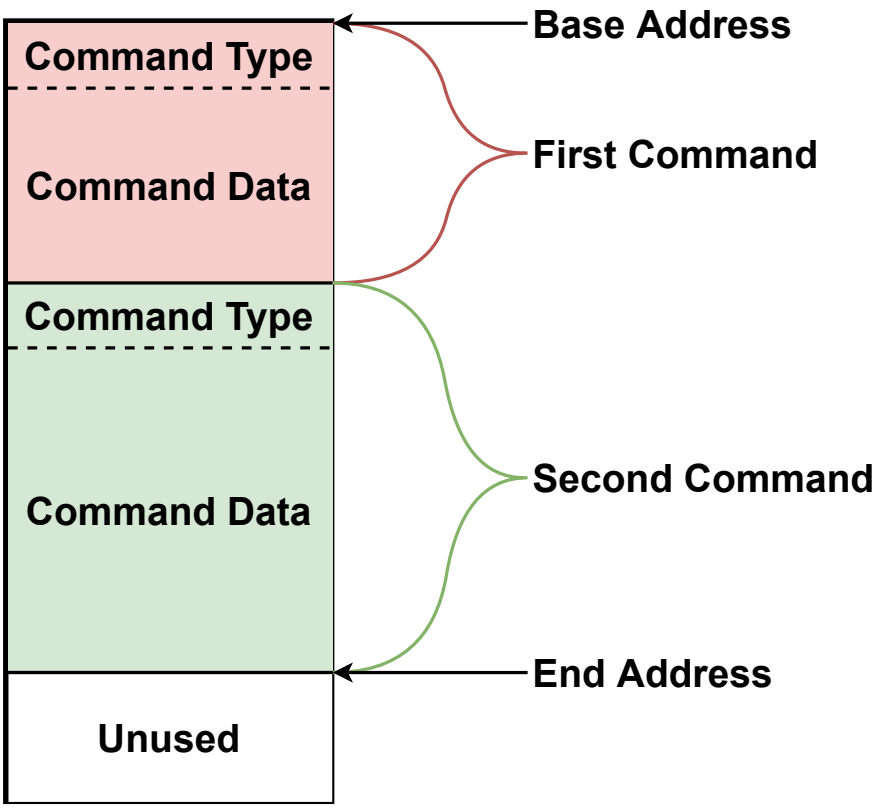


Figure 5.1: Data layout of the rendering command queue.

5.3 Renderer

Each command in the render queue can have its own set of rendering passes. In the following section, we will focus on scene rendering, ignoring elements like UI and debug visualization.

5.3.1 Overview

A scene is divided into static and dynamic parts. The static part is the world geometry, and the dynamic part consists of all the lights in the scene, which are all spotlights. The renderer is responsible for shading a scene by solving the reflectance equation [Kaj86], where our Bidirectional Reflectance Distribution

Function (BRDF) [Nic65] choices are Lambertian [Lam60] for diffuse and Phong [Pho75] for specular. For direct lighting, the visibility function is handled with shadow mapping [Wil78]. We reduce shadow aliasing by either using Percentage Closer Filtering (PCF) [RSC87; Mic] or Percentage Closer Soft Shadows (PCSS) [Fer05]. Indirect lighting is done by voxel cone tracing over the hemisphere and accumulating the indirect diffuse part. By tracing along the reflection vector, we gather the indirect specular contribution. The voxel emittance data only contains the diffuse component of the BRDF. Multi-bounce specular is not supported in our renderer but it could be achieved using other data structures, as will be discussed in Section 8.1.

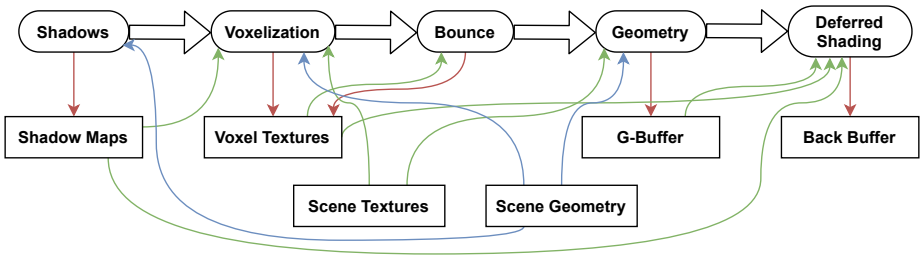
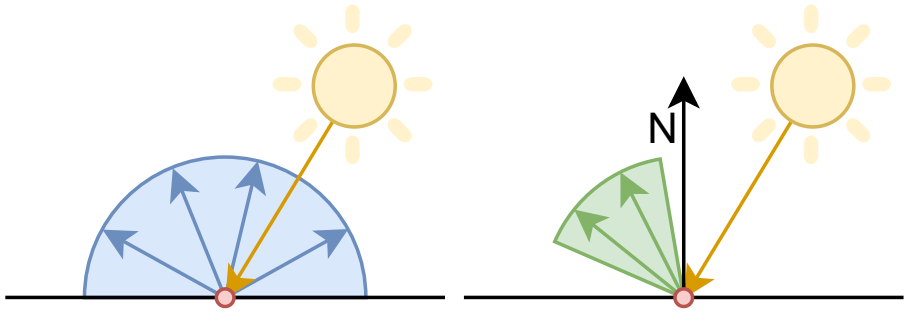


Figure 5.2: Timeline of GPU rendering operations. Arrows are color-coded: red for texture writes, green for texture reads, and blue for geometry reads.

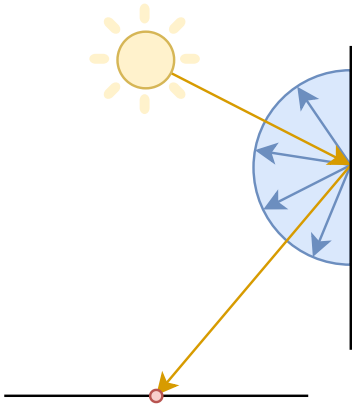
In the shadow pass, we draw a depth-only version of the scene to a shadow map for each light. The voxelization pass renders the scene into voxel textures containing opacity, diffuse emittance, and normals. The bounce pass computes a single light bounce by voxel cone tracing a voxel texture and outputs the result to a voxel texture. The geometry pass renders the scene from the camera's perspective and writes common attributes such as albedo, specular, normals, etc., into a set of geometry buffers. In the deferred shading pass, we use the G-buffer data to render direct lighting using shadow maps for the visibility function and cone tracing for indirect diffuse, indirect specular, and ambient occlusion. For shadows, we provide implementations of PCF and PCSS.

In our renderer, we have a separation for diffuse and specular as well as direct and indirect, see Figure 5.3. In our case, all direct lighting can be computed with closed-form analytical formulas and is therefore computationally cheap. Indirect diffuse is handled by voxel cone tracing and no solution for indirect specular is provided. However, indirect diffuse contributes to a local fragment's specular response, thereby providing reflections without specular highlights. Figure 5.4 illustrates the types of indirect lighting contributions in our renderer.

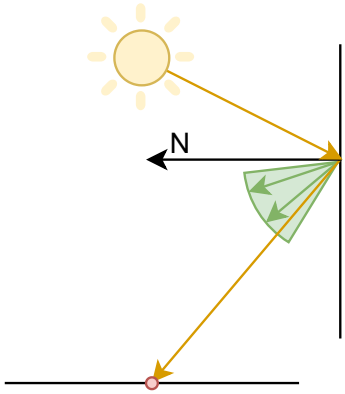


(a) Direct diffuse.

(b) Direct specular.



(c) Indirect diffuse.



(d) Indirect specular.

Figure 5.3: The four main components of light from our renderer's perspective.

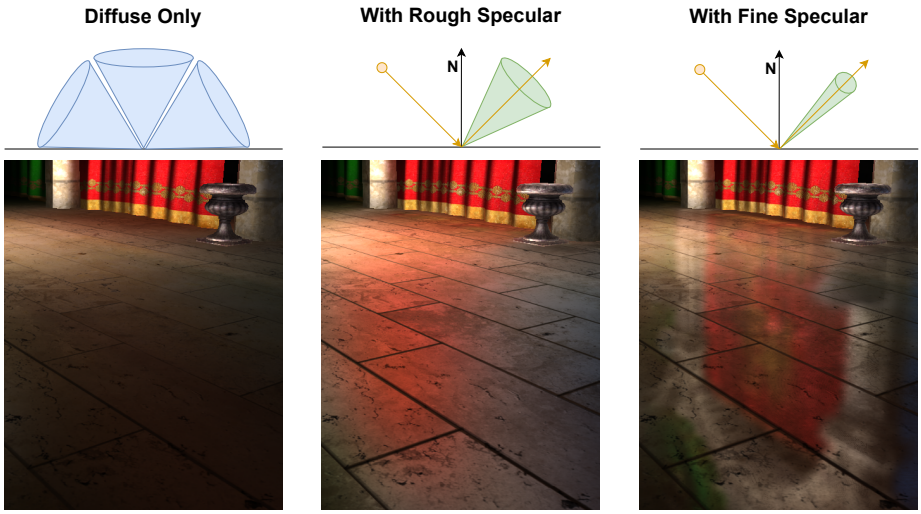


Figure 5.4: Types of indirect lighting contribution gathered from the deferred pass.

To assess the performance of the different stages in our renderer, we use Direct3D 11 queries [Mic21b]. To measure the amount of time needed to execute a series of commands, whether they are draw calls or compute dispatches, we must issue three queries in total. We issue timestamp queries before and after the group of commands. By subtracting the timestamp values, we obtain an elapsed number of cycles. Since we don't know the frequency at which the GPU ran during our measurement, we need another query over the command range to know the frequency and ensure that the results are valid. When valid, we store the results into a ring buffer containing the last 32 readings from which we compute the median value.

The application, including the renderer, is entirely single threaded. We therefore only use a Direct3D 11 immediate context to queue work for the GPU.

To both simplify development and improve performance, we use two structures to describe the entire pipeline state for graphics and compute. Instead of using the Direct3D 11 API directly to bind and unbind resources, we declare the entire state needed for a draw or compute operation up front and forward that to a single function of our own that sets all the state accordingly through Direct3D 11. The function only issues API calls upon state changes, thus avoiding CPU overhead due to the API validating the inputs. This design is mostly inspired by the newer explicit APIs, namely Vulkan with its Pipeline State Objects [Vul21] and Direct3D 12 with its Pipeline State [Mic21a]. Unlike these, we do not create immutable objects representing the pipeline state but only have the descriptor.

Because we have integrated third-party code that uses the Direct3D 11 API and not our wrapper, we need and use a way to invalidate the stored pipeline state. When the state is marked as invalidated, the state setting function doesn't need to compare anything and just sets all the state through the Direct3D.

5.3.2 Inverted Depth Buffer

First and foremost, we need to specify what depth buffer encoding we want to use in our renderer. Shadowing and lighting is affected by the accuracy of the data stored in the depth buffer. All depth formats are not equal, and therefore it's crucial to choose the right one for precision and performance. We seek to minimize the maximum error across the depth range while keeping memory bandwidth low. If we use a floating-point depth buffer and set the near clip plane at 0 and the far clip plane at 1, then we have two factors contributing to precision loss away from the camera. The first one is the z/w encoding of the fragments' depth, and the second one is the floating-point encoding itself. If we pit both of these against each other, they mostly cancel out. This is achieved by swapping the near and far clip plane depth values, i.e. the near clip plane is at 1, and the far clip plane at 0. Traditionally, this type of depth buffer is known as inverted depth or reversed depth.

Pettineo [Pet10] did a survey of the different kinds of depth buffer formats, which we used as a reference for choosing the one used in our implementation. He took a series of objects and placed them at random depths. Then he compared the position reconstructed from the depth buffer with a position interpolated from the vertex shader, calculating the error between the two, as can be seen in Figure 5.5.

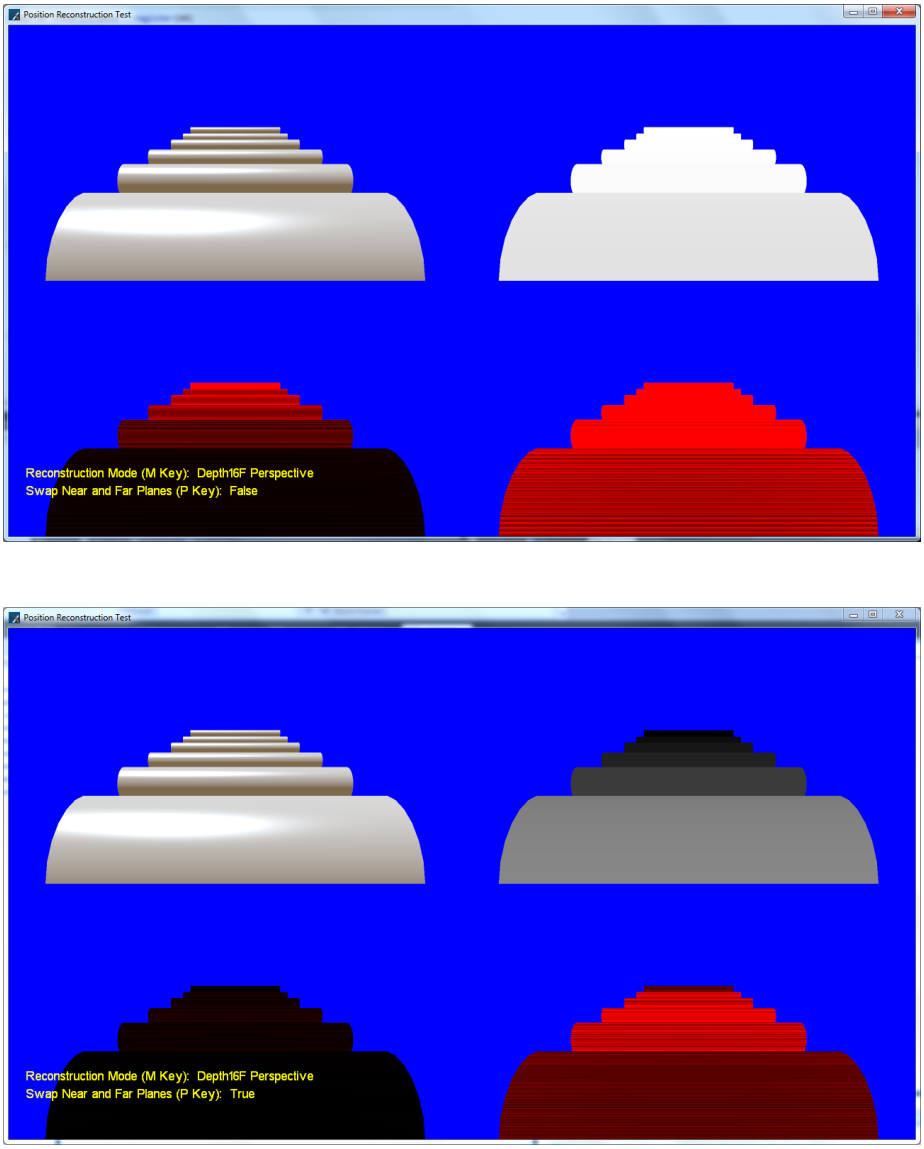
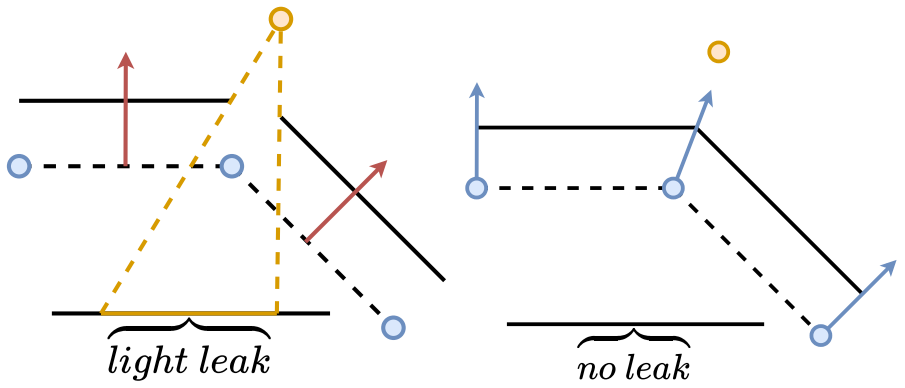


Figure 5.5: Depth buffer survey by Pettineo [Pet10] showing the difference between (Top) 16-bit float non-inverted and (Bottom) 16-bit float inverted. Note that for each image, the top left quadrant shows the rendered image, the top right shows the depth texture, the bottom left shows the error, and the bottom right shows the error multiplied by 100. Image courtesy of Pettineo [Pet10].

5.3.3 Shadows

To reduce the overall complexity of our solution, we implemented spotlights only. For each spotlight, only one shadow map is needed, where we draw the scene from the light's point of view using a perspective projection.

We apply a depth bias when writing to the shadow map, instead of when reading and comparing. That final bias is itself the sum of two biases: a constant value and a normal offset bias [Hol11]. In our implementation, we use the vertex normal instead of the face normal because it helps prevent the generation of small cracks in the shadows map, as illustrated in Figure 5.6.

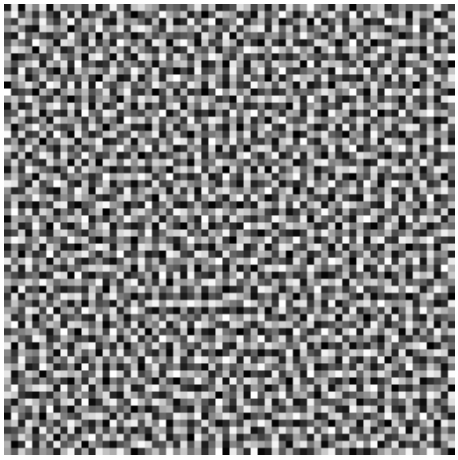


(a) Triangles are offset by face normals, creating gaps and therefore light leaks.

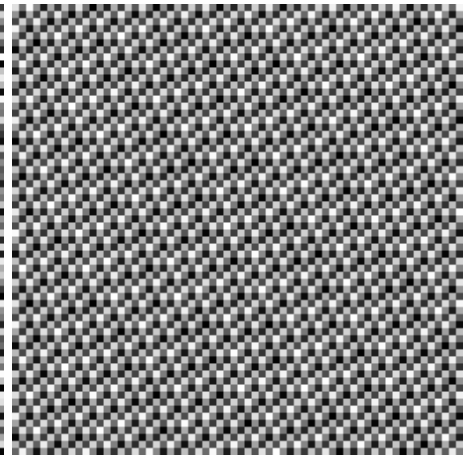
(b) Triangles are offset by vertex normals.

Figure 5.6: (Left) Face normal offset. (Right) Vertex normal offset.

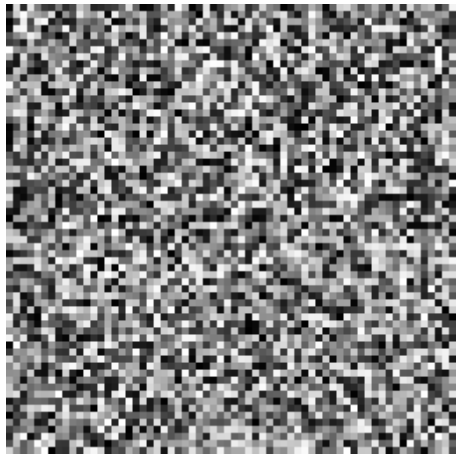
We have two filtering modes available. The first one is PCF, and has been implemented using a hardware comparison sampler (Figure 5.8a). The second one is PCSS, with Poisson disc sampling (Figure 5.8b & Figure 5.8c). The blocker search uses 16 Poisson disc samples and the filter uses 16 randomly rotated Poisson disc samples. The disc is rotated by a random angle computed with interleaved gradient noise [Jim14] (IGN) on the fragments' world-space positions. Listing 3 shows how we generate the rotation matrix. Ideally, we would use blue noise but it is expensive to generate on-the-fly and the alternative involves reading data from a texture. IGN provides a lower quality noise, but at an extremely low cost while being temporally stable. See Figure 5.7 for illustrations of different kinds of noise.



(a) Blue noise.



(b) Interleaved Gradient Noise.



(c) White noise.

Figure 5.7: Comparison of the considered noise patterns.

(a) Hardware Percentage Closer Filtering (4 depth values).

(b) Percentage Closer Soft Shadows with 16 search taps (no PCF) and 16 Poisson disc filter taps (with PCF).

(c) PCSS with the same settings, but the Poisson disc is randomly rotated by an angle generated with IGN.

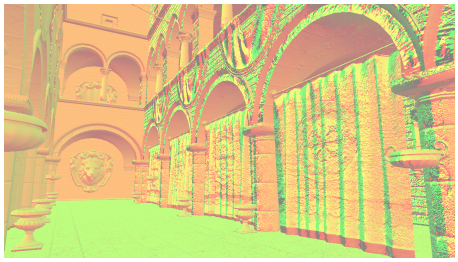
Figure 5.8: Comparison of shadow map filtering methods. Our application implements methods **(a)** and **(b)**.

```
// Interleaved Gradient Noise by Jorge Jimenez
// from "Next Generation Post Processing in Call of Duty: Advanced Warfare"
float InterleavedGradientNoise(float2 uv)
{
    float3 magic = float3(0.06711056, 0.00583715, 52.9829189);
    return frac(magic.z * frac(dot(uv, magic.xy)));
}

// We extend IGN to hash a 3-dimensional input.
float InterleavedGradientNoise(float3 uv)
{
    float ign = InterleavedGradientNoise(uv.xy);
    return InterleavedGradientNoise(float2(ign, uv.z));
}

float2x2 CreateRandomRotationMatrix(float3 position)
{
    float angle = InterleavedGradientNoise(position) * 2.0 * PI;
    float sin, cos;
    sincos(angle, sin, cos);
    return float2x2(cos, -sin, sin, cos);
}
```

Listing 3: Our solution for generating a random 2D rotation matrix from a 3D vector. In our use case, the 3D vector is the world-space position of the fragment being shaded.

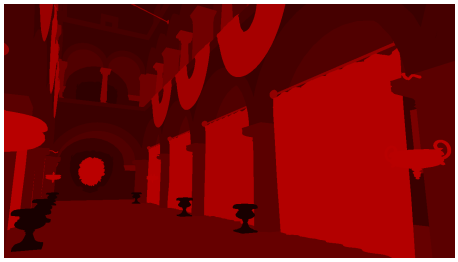
5.3.4 Geometry Pass

The geometry pass prepares data for the deferred shading pass. For an overview of forward and deferred shading, see Chapter 6. It writes per pixel information about the depth, surface normal, world-space position, color, etc. to geometry buffers (G-buffers), see Figure 5.9. The data written depends entirely on the shading needed in the deferred pass.

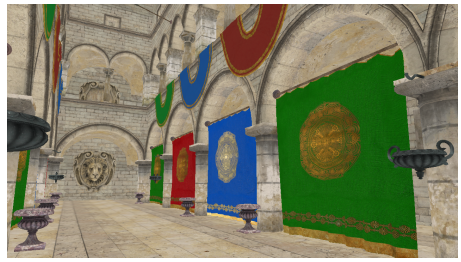


(a) Surface normals

(b) World-space position



(c) Material id



(d) Albedo color

(e) Specular reflectance intensity

Figure 5.9: Images highlighting different per pixel data being rendered to g-buffers.

Note that the world-space position buffer is redundant since we already have the hardware depth buffer. By back-projecting the depth value back into world-space, we can obtain the position, and save a considerable amount of memory as well as memory bandwidth. See Figure 5.10 for the storage size of the different attributes.

		Channel Usage				
Texture Format	BPP	R	G	B	A	Notes
R8G8B8A8_UNORM	32	Albedo			Specular	A is the specular reflectance's intensity
R16G16_SNORM	32	World-Space Normal Vector		N/A		Octahedral encoding
R8_UINT	8	Material Index	N/A			
R32_FLOAT	32	Depth (z/w)	N/A			Hardware depth buffer, no stencil buffer used
R32G32G32A32_FLOAT	128	World-Space Position			Unused	For convenience (can be derived from depth)

Figure 5.10: Breakdown of our G-buffer. BPP is short for bits per pixel.

We use octahedral encoding for packing the surface normal in two 16-bit unorm integers, as described by the oct32 encoding [Cig+14]. The octahedral encoding maps the octants of a sphere to the faces of an octahedron, which it then projects onto the plane and unfolds into a unit square, as seen in Figure 5.11.

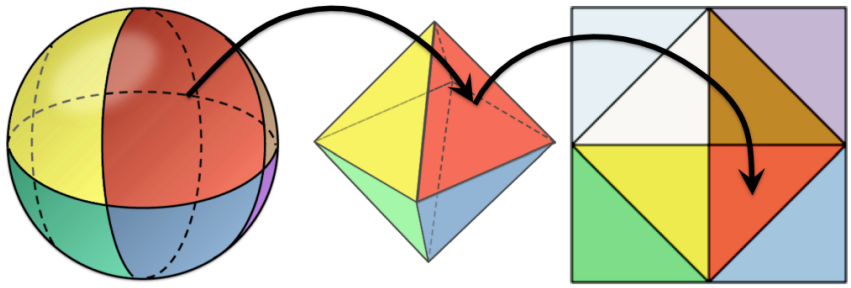


Figure 5.11: How the octants of a sphere are mapped to the faces of an octahedron, which are then projected onto the plane and unfolded into a unit square. Image courtesy of Cigolle et al. [Cig+14].

We also do normal mapping as a part of our geometry pass. Traditionally, there are two choices, the first of which stores the tangent vector as a vertex attribute and reconstructs the bitangent vector in one of the GPU's programmable shader stages. The second option is to generate both on-the-fly. In our implementation, we went with the second choice as described by Schüler [Sch13] on his website.

5.3.5 Direct Lighting

Since we use the Sponza data set from McGuire [McG16], the materials are modeled for a simplistic shading model: Lambertian diffuse and Blinn's interpretation of Phong's specular model. In our case, we use the Lambertian diffuse and a similar version of Phong's specular model. We use this BRDF:

$$f = (1 - F)K_d + FK_s max(\mathbf{r} \cdot \mathbf{v}, 0)^{N_s}, \tag{5.1}$$

where F is the Fresnel reflectance, K_d is the diffuse color, K_s is the specular color, $\mathbf{r}$ is the reflection vector, $\mathbf{v}$ is the view vector, and N_s is the specular exponent.

To compute the Fresnel reflectance, we use Schlick's approximation [Sch94] as follows:

$$F(\mathbf{n}, \mathbf{l}) = F_0 + (1 - F_0)(1 - max(\mathbf{n} \cdot \mathbf{l}, 0))^5, \tag{5.2}$$

where F_0 is the characteristic specular color or the fraction of light reflected when the light is perpendicular to the surface, $\mathbf{n}$ is the normal to the surface, and $\mathbf{l}$ is the light vector.

To compute direct lighting, we iterate over every light, evaluate the BRDF, light intensity as well as visibility through shadow map look-ups and combine the results:

$$L_o = \sum_{i=0}^{N} f I_i V_i max(\mathbf{n} \cdot \mathbf{l}_i, 0), \tag{5.3}$$

where N is the number of lights, I_i is the unshadowed incoming radiance (which we called light intensity above), V_i the visibility function, $\mathbf{n}$ is the surface normal, $\mathbf{l}_i$ the light vector. $I_i V_i$ is the incoming radiance L_i from the reflectance equation.

5.3.6 Voxel Cone Tracing

5.3.6.1 Overview

In our implementation of voxel cone tracing global illumination, we leverage the graphics pipeline to voxelize the scene with one draw call per material. We support hardware conservative rasterization when available and provide a software implementation of our own as a fallback. The attributes are stored in a dense grid of anisotropic voxels with no cascade or clip-map scheme. We chose dense voxel grid storage since it is the simplest option. However, cascades would have been the better choice if more time was available. Instead of only rendering emissive surfaces during the voxelization step and injecting direct diffuse light with the help of reflective shadow maps [DS05] in subsequent steps, we render all direct diffuse lighting as part of the voxelization step itself. To do this, we evaluate direct diffuse lighting in the pixel shader by iterating over every light in the scene and computing the visibility function using PCF shadows and the incoming radiance from the light and evaluating the reflectance equation with a Lambertian BRDF. The pre-filtered scene data in the mip-map pyramid is generated with the help of compute dispatches. We launch the final gather for indirect lighting in the deferred shading pass. If we evaluate more than one light

bounce, we run a compute dispatch prior to deferred shading that traces the voxelized scene and outputs a new voxelized version.

5.3.6.2 Voxelization

Voxelization is the process of taking an input scene and creating a voxelized representation. In the voxelization pass, we generate a voxelized representation of opacity and emittance, see Figure 5.12. The opacity stores what fraction of light gets blocked going in through a voxel face, 1.0 being fully solid (no light gets through) and 0.0 being empty (all light gets through). The emittance stores what amount of light comes out through a voxel face.

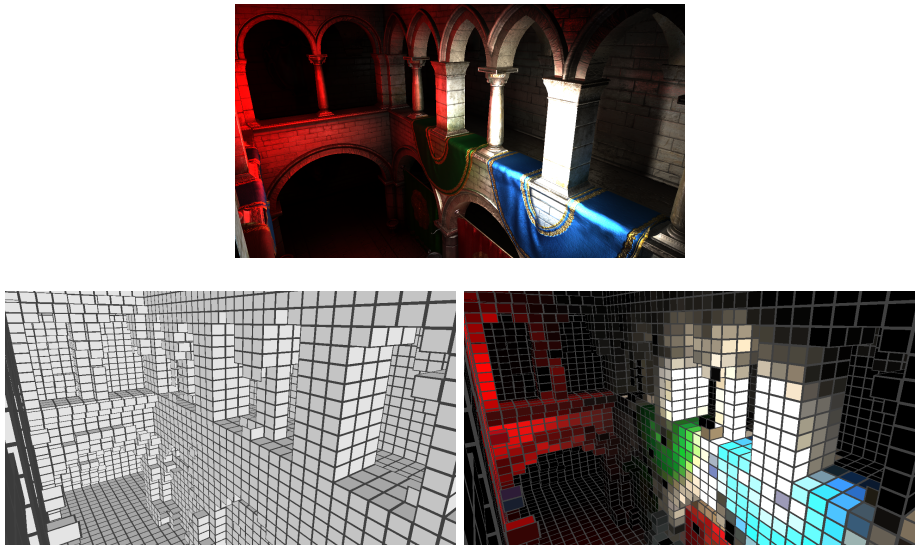


Figure 5.12: (Top) The scene. (Bottom Left) Opacity. (Bottom Right) Emittance.

When voxelizing the scene's opacity or emittance, we send the needed attributes to vertex shaders specialized for each case. The data gets forwarded to the geometry shader, where we do the dominant axis selection and optionally the first half of the software conservative rasterization algorithm we selected which inflates the triangle. Recall that conservative rasterization is used to generate fragments when any part of the pixel is covered, not just the sample location. There is hardware support for conservative rasterization on newer graphics cards. We created a software implementation for cards that don't support hardware rasterization, which we present later in this chapter.

Finally, in the pixel shader, we do the second phase of conservative rasteriza-

tion which is fragment clipping. Here we reject the pixel if it lies outside of the axis-aligned bounding box computed in the geometry shader. If the pixel is within bounds, our job is to compute the final voxel attributes needed for output. It's important to note that the output isn't a render target but rather an unordered access view (UAV). Furthermore, we always write three faces, as we use anisotropic voxels.

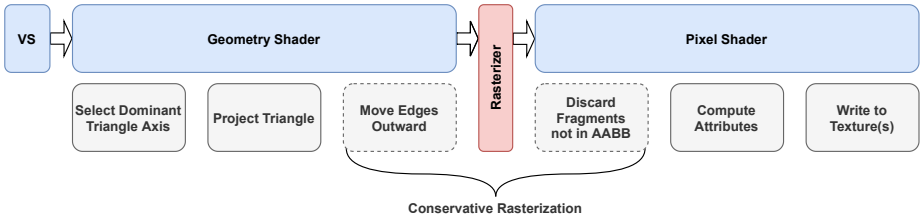


Figure 5.13: The voxelization pipeline. Programmable GPU stages in blue. Fixed function stages in red.

We use shader atomics for writing to some of these UAVs. An atomic operation allows multiple threads involved in the GPU to communicate using group shared memory. Specifically, we use them to compute averages or max values for a voxels attribute, depending on the HDR mode. The details will be covered in Section 5.3.6.5.

There is a small additional optimization. The viewport is selected based on the dominant axis by using an HLSL semantic in the geometry shader. With this, we can avoid generating too many fragments for certain axes when using a mixed resolution voxel grid.

For software conservative rasterization, we define planes passing through the edges of the projected triangle and intersecting the origin, and push them outward along their normal vector. We then calculate the intersections, and get the final position as described by Hasselgren et al. [HAO05]. See Figure 5.14 for an illustration.

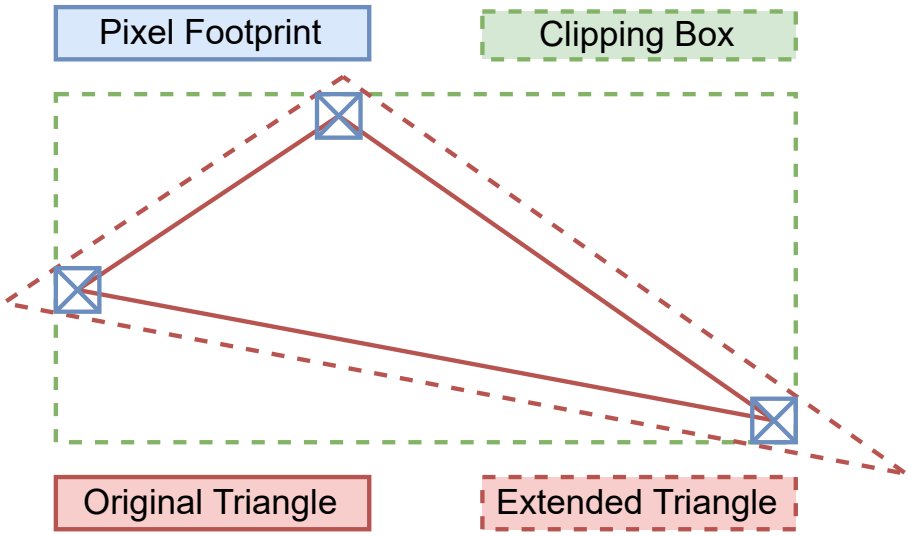


Figure 5.14: The input triangle (solid red) has its edges extended outward (dashed red) by the worst-case semi-diagonal (blue). Because the new triangle extends too far, it is clipped against a bounding box (green).

The final steps of voxelization are computing the final attribute values and storing them into our voxel data structure. For opacity we simply store a full opacity value for each fragment (no averaging). For emittance we compute direct diffuse lighting and write out an averaged result, which is done in different ways depending on the HDR mode. Finally, we average the normal and write it as well. As we use anisotropic voxels, we need to get the right offset into our data structure, which we get from the normal. Finally, we always write to three voxel faces.

As will be covered in Section 5.3.6.7, we generate mip-maps using front-to-back compositing to set up the data for tracing.

5.3.6.3 Anisotropic Voxels

For a given memory budget, one can opt for isotropic voxels or instead opt for anisotropic voxels, (Figure 5.15) but with a grid whose resolution is two times lower on all axis. That is, the increased storage cost (6x because there are 6 faces per voxel) has to be compensated by a lower resolution (8x). We went with anisotropic over isotropic, as it provides better colors and less light leaking for a slightly lower memory usage. The disadvantage is that the trace results are blurrier.

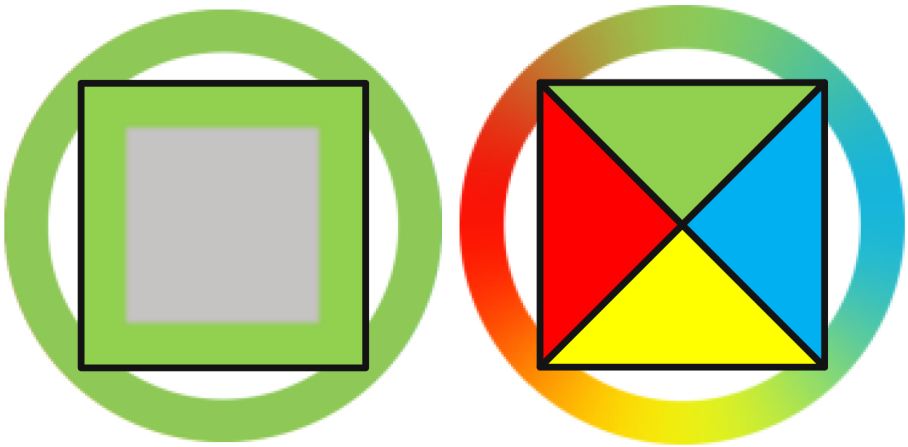


Figure 5.15: (Left) Isotropic voxel representation. (Right) Anisotropic voxel representation. Images courtesy of Mittring [Mit12].

The maximum 3D texture dimension is 2048 at all feature levels, even for Direct3D 12. Since we pack the attribute data for all 6 faces in a single texture, the maximum power of two dimension available to us is therefore 256 for one of the three axis. To enable the use of a higher resolution grid, we can store the face attributes in 6 separate 3D textures and dynamically index the texture array from the shader. However, dynamic texture array indexing requires HLSL 5.1 support and is only available through Direct3D 12 (Listing 4). For Direct3D 11, we are forced to create a branch for each texture access if we use an array of 3D textures (Listing 5). Instead we store the 6 faces in one texture (Listing 6, Figure 5.16). In all cases, the faces are stored in the following order: $+x$, $-x$, $+y$, $-y$, $+z$, $-z$.

```
// Texture dimensions: X, Y, Z
Texture3D voxelGrid[6] : register(t0);

void WriteToVoxel(uint3 tc, float3 normal, float4 value)
{
    uint3 faceOffsets = normal >= 0.0 ? 1 : 0;
    uint3 faceIndexes = uint3(0, 2, 4) + faceOffsets;
    voxelGrid[faceIndexes.x][tc] = value;
    voxelGrid[faceIndexes.y][tc] = value;
    voxelGrid[faceIndexes.z][tc] = value;
}
```

Listing 4: HLSL 5.1 - dynamic indexing into an array of textures.

```
// Texture dimensions: X, Y, Z
Texture3D voxelGrid[6] : register(t0);

void WriteToVoxel(uint3 tc, float3 normal, float4 value)
{
    if(normal.x < 0.0)
        voxelGrid[0][tc] = value;
    else
        voxelGrid[1][tc] = value;

    if(normal.y < 0.0)
        voxelGrid[2][tc] = value;
    else
        voxelGrid[3][tc] = value;

    if(normal.z < 0.0)
        voxelGrid[4][tc] = value;
    else
        voxelGrid[5][tc] = value;
}
```

Listing 5: HLSL 5.0 - branching access into an array of textures.

```
// Texture dimensions: 6*X, Y, Z
Texture3D voxelGrid : register(t0);

void WriteToVoxel(uint3 tc, float3 normal, float4 value)
{
    uint3 texDim;
    voxelGrid.GetDimensions(texDim.x, texDim.y, texDim.z);
    texDim.x /= 6;
    uint3 faceOffsets = normal >= 0.0 ? 1 : 0;
    uint3 faceIndexes = uint3(0, 2, 4) + faceOffsets;
    uint3 tcOffsets = texDim.x * faceIndexes;
    voxelGrid[tc + uint3(tcOffsets.x, 0, 0)] = value;
    voxelGrid[tc + uint3(tcOffsets.y, 0, 0)] = value;
    voxelGrid[tc + uint3(tcOffsets.z, 0, 0)] = value;
}
```

Listing 6: HLSL 5.0 - branchless access into an single texture.

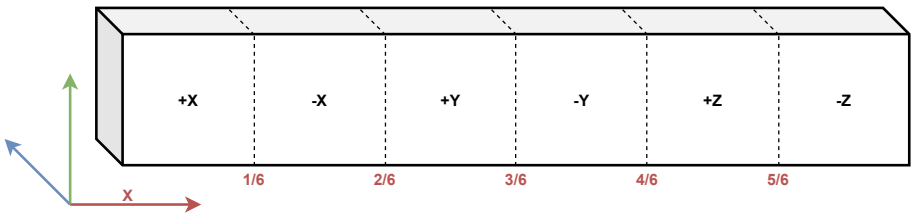


Figure 5.16: Single texture storing all six faces.

We clamp texture coordinates to avoid data bleeding through face boundaries, since we use trilinear texture filtering during tracing. For each face, the box we clamp to is the face's box shrunk by half the size of a voxel on the x axis.

5.3.6.4 Voxel Attribute Storage

The voxels encode opacity, surface normals, and emittance, each with their own format. Opacity is stored as one byte per voxel face and doesn't have any variations in relation to the quality modes. Opacity is utilized for an AO-only rendering mode, where the opacity data structure is traced using voxel cone tracing.

When we store emittance, there are three quality modes options: LDR, CHDR, and HDR. Low dynamic range (LDR) computes the moving average and writes to a low bit depth texture. Compressed high dynamic range (CHDR) stores the maximum value into a low bit depth texture using a custom encoding to increase the value range. This gets expanded and written to a high bit depth texture in a follow-up compute shader pass. High dynamic range (HDR) stores the emittance sums and fragment counter into three separate textures. It divides the sum by the counter in a subsequent compute shader pass to get the average. See Figure 5.17 for a breakdown of LDR, CHDR, and HDR voxelization and fix-ups.

All three quality modes rely on atomic operations to write to their UAVs. Without them, a voxel's final value is that of the last fragment written. Since the order is not deterministic, the final value can change drastically from frame to frame, thereby causing temporal instability. Because we compute indirect lighting at a very regular interval, this is needed to prevent indirect lighting contributions from flickering too much. All voxelization UAVs are 32-bit textures because its the only bit depth supported for atomic operations on textures in Direct3D 11. Otherwise, the HDR mode would simply write to a single 64-bit texture directly.

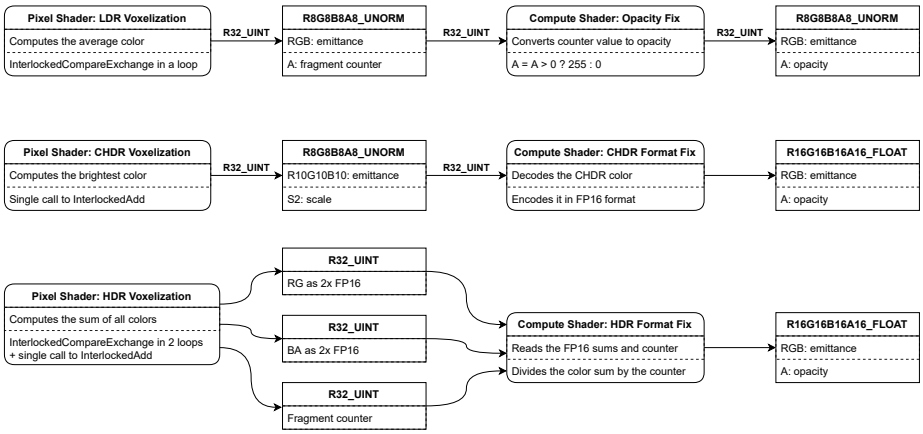


Figure 5.17: Emittance voxelization and fix-up breakdown for all three quality modes. Texture formats on arrows going out of shader stages represent UAV formats and formats going into shader stages represent SRV formats. If an arrow has no format name, then the view uses the format the texture was created with.

The surface normal attribute is encoded similarly to LDR emittance with one byte per channel, where the alpha channel is used as a fragment counter to compute the moving average using shader atomics. Unlike the surface normal, the opacity attribute is written without the use of any shader atomics because we only support fully opaque surfaces. Opacity and normal have a fixed cost relative to the emittance quality mode. Figure 5.18 shows the memory usage scaling for the different quality modes and number of bounces.

	Memory Usage of a 128x64x64 Voxel Grid, in MB							
	Opacity	Normal	Emittance 1B	Emittance 2B	Emittance 3+B	Total 1B	Total 2B	Total 3+B
LDR	3	12	13	27	41	29	42	56
CHDR	3	12	41	68	95	56	83	111
HDR	3	12	68	95	123	83	111	138

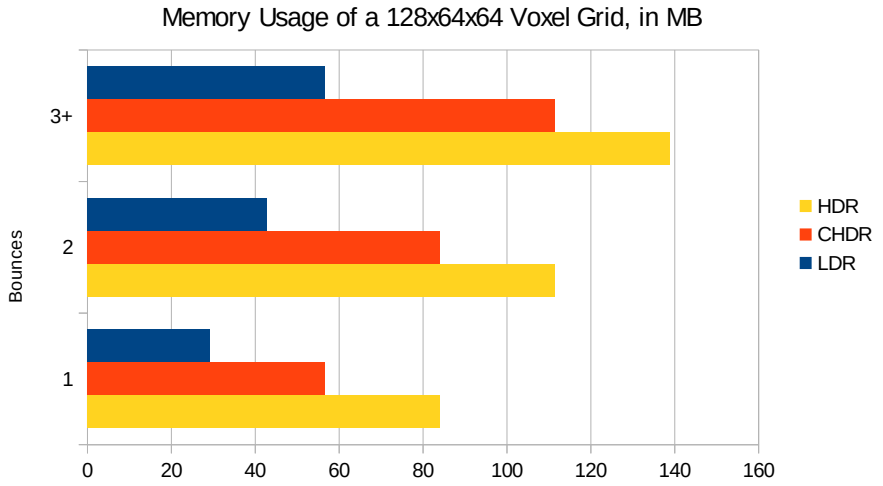


Figure 5.18: Memory usage breakdown for a 128x64x64 voxel grid.

5.3.6.5 Atomic Operations

For our low bit depth unorm formats, storing a sum of fragment values (LDR emittance, normals) is not doable because the values would overflow very quickly. An alternative would be to store the individual fragments to average in a linked list or a fixed size array in some intermediate buffers and textures. The memory cost of this for a fragment count high enough to ensure temporal stability would be prohibitive. Instead, we use a moving average scheme with the help of atomic operations. To compute a moving average, only two values needs to be stored: the average and the counter. Both values start at 0 and whenever a new value is added to the mix the counter is increased and the new average is computed as follows:

$$a = \frac{a\ c + v}{c + 1}, \tag{5.4}$$

where a is the average, c is the counter, and v is the new value.

For the implementation, the idea is to compute the new value and loop on a compare and swap (CAS) operation, also known as compare-exchange or InterlockedCompareExchange in HLSL, until the expected value, from which we have computed the new value, has not been changed by another thread. The function

starts by expecting the current counter and average to be 0 as the UAVs are cleared to 0 before the voxelization draw call is issued. It starts by computing a new average value based on that assumption, meaning the new average is the value provided, and the new counter is 1. Then, we loop on CAS until it either succeeds, the counter is about to overflow, or we ran too many iterations. The last condition is there to help improve performance at the cost of image quality and temporal stability since it discards fragments randomly. When the compare-exchange operation lets us know that the stored value is different from the one we expected, we have to recompute the average and counter values and use that new value from the compare-exchange as the new expected value for

the next loop iteration:

```
uint PackUnorm4x8U(float4 input)
{
    uint4 bytes = uint4(input);
    return (bytes.w << 24u) | (bytes.z << 16u) | (bytes.y << 8u) | bytes.x;
}

float4 UnpackUnorm4x8U(uint input)
{
    uint4 bytes = uint4(input, input >> 8u, input >> 16u, input >> 24u) &
    ↪  0xFFu;
    return float4(bytes);
}

void AtomicAverage(RWTexture3D<uint> dest, uint3 tc, float3 value, uint
↪  maxIterations)
{
    value = saturate(value) * 255.0; // denormalize the input
    // we start by expecting the current fragment to be the first,
    // i.e. the expected counter is 0 and the new one is 1
    uint newValue = PackUnorm4x8U(float4(value, 1.0));
    uint expValue = 0; // 0 value (black color, 0-length direction) and 0
    ↪  counter
    for(uint i = 0; i < maxIterations; ++i)
    {
        uint curValue;
        InterlockedCompareExchange(dest[tc], expValue, newValue, curValue);
        if(curValue == expValue || (curValue >> 24) == 255)
        {
            // we stop here when we either successfully wrote our new color
            // or have been trying too many times (else the counter
            ↪  overflows)
            break;
        }

        float4 curValueF = UnpackUnorm4x8U(curValue);
        float4 newValueF;
        newValueF.rgb = curValueF.rgb * curValueF.a;
        newValueF.rgb += value.rgb;
        newValueF.rgb /= curValueF.a + 1.0;
        newValueF.a = curValueF.a + 1.0;
        newValue = PackUnorm4x8U(newValueF);
        expValue = curValue;
    }
}
```

Listing 7: Atomic moving average for a R8G8B8A8 unorm texture. The R32 uint view is an API requirement for the usage of atomic operations.

For HDR emittance, we store the sum of the RG and BA values separately into two textures and the fragment counter in a third texture, as illustrated

in Figure 5.17. As the fragment counter is a single value, the addition can be done atomically using the InterlockedAdd HLSL intrinsic directly. For the color values, we rely on the CAS operation again:

```
void AtomicAddFloat2x16(RWTexture3D<uint> dest, uint3 tc, float2 value, uint
↪  maxIterations)
{
    uint newValue = PackHDR(value);
    uint expValue = 0;
    for(uint i = 0; i < maxIterations; ++i)
    {
        uint curValue;
        InterlockedCompareExchange(dest[tc], expValue, newValue, curValue);
        if(curValue == expValue)
        {
            break;
        }

        float2 expValueF = UnpackHDR(curValue);
        float2 newValueF = value + expValueF;
        newValue = PackHDR(newValueF);
        expValue = curValue;
    }
}

void AtomicAddHDREmittance(RWTexture3D<uint> rg, RWTexture3D<uint> ba,
↪  RWTexture3D<uint> counter, uint3 tc, float3 value)
{
    AtomicAddFloat2x16(rg, tc, value.rg, uint(-1));
    AtomicAddFloat2x16(ba, tc, float2(value.b, 1.0), uint(-1));
    InterlockedAdd(counter[tc], 1);
}
```

Listing 8: Atomic addition of two 16-bit floating point values packed in a 32-bit uint texture.

The performance is not as optimal as it could be since we have effectively no maximum iteration count. We can't simply set a maximum iteration count without checking how AtomicAddFloat2x16 returns because if one of the two calls to it reaches the maximum iteration count, we would still proceed to execute the InterlockedAdd intrinsic and our sums and counters will be out of sync. A solution would be to test the return condition of the first call and return if the maximum number of iterations was reached. That way, none of the textures is written to.

For the CHDR emittance format, we store the maximum value using the InterlockedMax intrinsic. For this to work, we store the scale in the high bits of the uint value, such that a max operation always retains the color with the highest scale. If the scale of two compared values is the same, the max operation will

not always select the brightest color but instead pick based on the individual values of the color channels. Because we store red, green, blue, scale in the low to high bit order, the max value would be selected based on the scale, blue, green, red values in order. Here are the packing and unpacking functions for our CHDR format:

```
uint PackCHDR(float3 input)
{
    float Sf = max3(input.r, input.g, input.b);
    Sf = ceil(Sf);
    Sf = clamp(Sf, 1.0, 127.0);
    float3 Cf = (input / Sf) * 255.0;
    Cf = clamp(Cf, 0.0, 255.0);
    uint3 Cu = uint3(Cf);
    uint Su = uint(Sf);
    uint result = Cu.r | (Cu.g << 8u) | (Cu.b << 16u) | (Su << 24u) | (1 <<
    ↪  31u);
    return result;
}

float3 UnpackCHDR(uint input)
{
    uint3 Cu = uint3(input & 0xFFu, (input >> 8u) & 0xFFu, (input >> 16u) &
    ↪  0xFFu);
    float3 Cf = float3(Cu) / 255.0;
    uint Su = (input >> 24u) & 0x7Fu;
    float Sf = float(Su);
    float3 result = Cf * Sf;
    return result;
}
```

Listing 9: Encoding and decoding of our 32-bit uint CHDR color format.

5.3.6.6 Conservative Rasterization

We have three rasterization modes available. The first one is non-conservative rasterization. The second one is software conservative rasterization, and has been implemented as described in Section 5.3.6.2. The final one is hardware conservative rasterization, and is only available when using Direct3D 11.3 on tier two capable hardware.

5.3.6.7 Mip Generation

Because voxel cone tracing is analogous to ray-marching through participating media, the pre-filtering process requires directional integration, much like front-to-back compositing. For a group of eight neighbor voxels, we compute the corresponding pre-filtered voxel in two steps. First, we apply front-to-back compositing to four groups of two voxels in the direction opposite to the face's

normal. Then, the four resulting values are averaged as seen in Figure 5.19.

To apply directional integration, we use the following formulas to evaluate opacity and emittance:

$$O_{filtered} = O_{front} + (1 - O_{front})O_{back}, \tag{5.5}$$

$$E_{filtered} = E_{front} + (1 - O_{front})E_{back}. \tag{5.6}$$

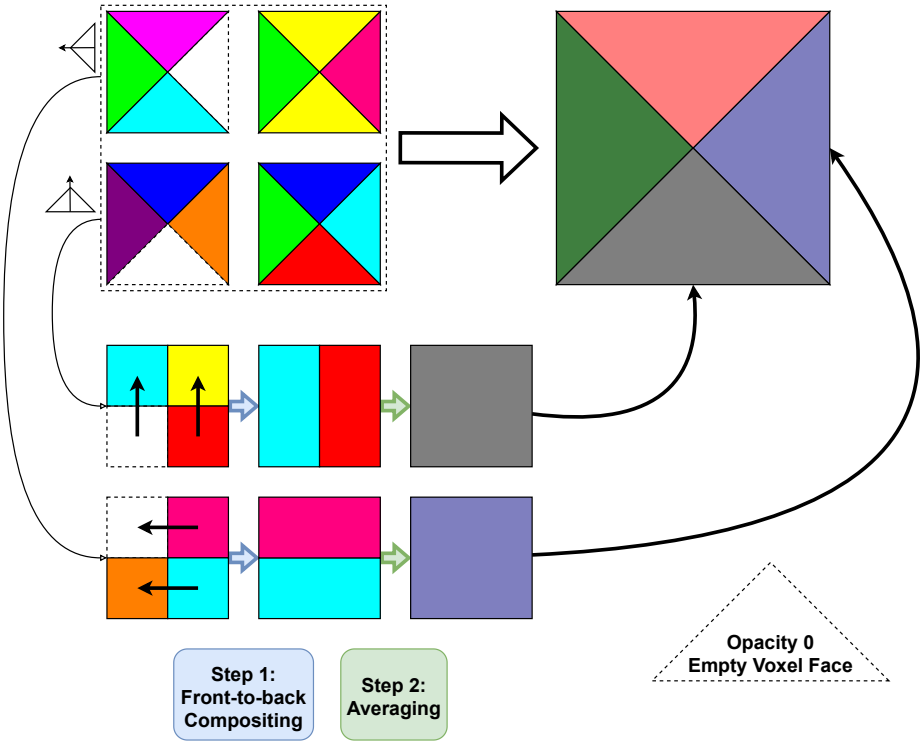


Figure 5.19: Pre-integration of four anisotropic voxels into one. The bottom two lines are detailed breakdowns of the process for the right and bottom faces. Dashed faces are transparent.

5.3.6.8 Voxel Cone Tracing

While ray-tracing [Whi79] intersects rays against the scene and yields collision points, cone tracing, also known as sphere tracing [Ama84], which we approximate with a varient of sphere tracing [Har96] presented by Crassin et al. [Cra+11],

intersects spheres of increasing dimensions (Figure 5.20) against the scene, yielding collision surfaces or volumes. Because of this, there needs to be a way to evaluate scene data over a given surface or volume. The geometry must therefore be stored in a pre-filtered form, which implies a precision loss due to averaging but reduces noise.

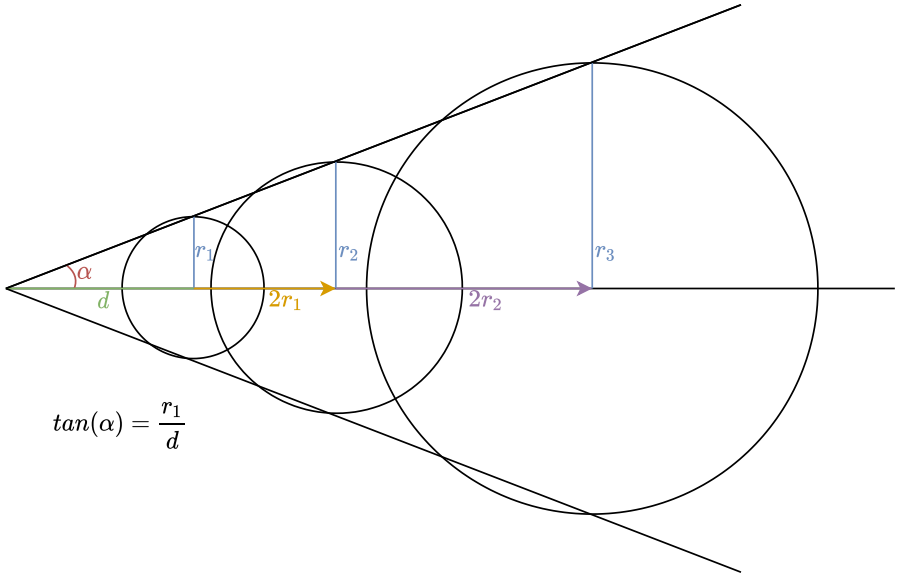
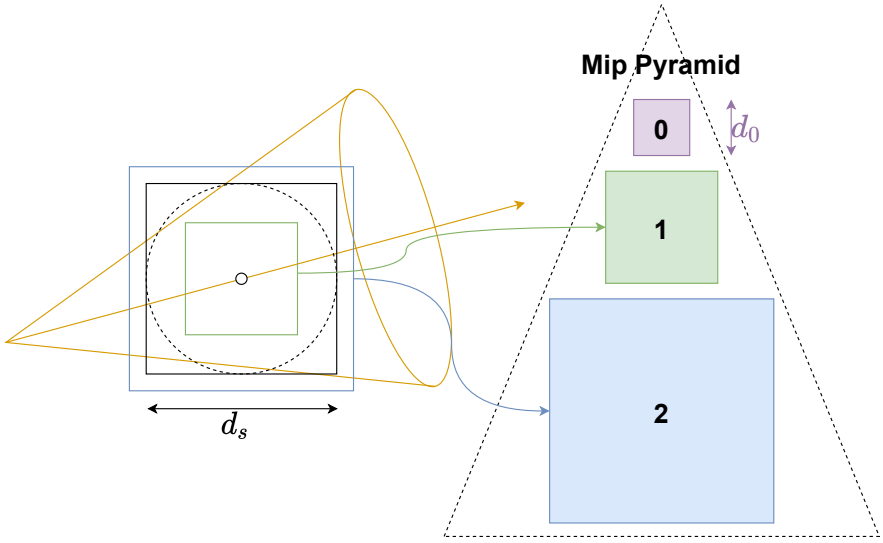


Figure 5.20: Cone tracing works by approximating the shape with consecutive spheres of increasing dimensions.

The axis-aligned voxel grid is stored in a 3D texture where pre-filtered versions of the scene are stored in its mip pyramid, all covering the same volume. Mip 0 is the most detailed unfiltered version of the scene, mip 1 is a pre-filtered version with half the resolution on all axis, mip 2 has a quarter of the resolution etc. In the trace, the dimension of the sphere is mapped to a mip level for rapid evaluation of scene attributes over the volume of interest, see Figure 5.21. For higher quality results, we interpolate between the two closest mip levels. Note that the pre-filtered data is stored in boxes (stretched cubes), not ellipsoids (stretched spheres).



$$mip = log2(\frac{d_s}{d_0}) = log2(3.4) \approx 1.765$$

$$mip_{low} = \lfloor mip \rfloor = 1$$

$$mip_{high} = \lceil mip \rceil = 2$$

Figure 5.21: The dashed circle represents the disk whose surface we're interested in evaluating. The black quad is the surface that can be evaluated by interpolating between the pre-filtered versions of the green and blue quads (the two closest mip levels).

A cone tracing function is given a starting position, orientation, and opening angle. In our case, the cone's orientation is a unit-length direction vector and its angle is stored as a coefficient computed as follows: $cone_{ratio} = 2tan(cone_{aperture}/2)$. The first sample location in the trace can never be the starting position because the surface we start from has rendered the voxel containing the starting position solid. The first sampling location must therefore be shifted forward in the direction of the trace to avoid self-intersection. For a given sample location, the sphere's diameter is: $cone_{ratio}\ sphere_{distance}$. To iterate to the next location, the current location is moved forward by the diameter of the current location's sphere. The trace stops when either the location is outside the scene or the accumulated opacity has reached a certain threshold, meaning that further contributions would not be significant enough to justify

the additional cost. Here's is an overview of the algorithm in code form:

```
float ConeTraceOpacity(
    float3 origin, float3 dir, float coneRatio,
    float stepScale, float opacityThreshold)
{
    // texture dimensions != grid dimensions
    float3 gridDim = GridDimensions();
    // shift first sampling location to avoid self-intersection
    float minDist = VoxelSizeMip0(dir);
    float dist = minDist;
    float stepLength = minDist; // uncorrected in first loop iteration
    float3 smpPos = origin + dir * dist;
    float accOpacity = 0.0; // accumulated opacity
    while(IsPointInScene(smpPos) && accOpacity < opacityThreshold)
    {
        // the max call prevents the step size from becoming too
        // small, which would lead to performance degradation
        float sphereDiameter = max(dist * coneRatio, minDist);
        float nextStepLength = sphereDiameter * stepScale;
        float lod = log2(sphereDiameter / minDist);
        float smpOpacity = SampleOpacity(lod, smpPos, dir);
        smpOpacity = CorrectOpacity(smpOpacity, stepLength / minDist);
        accOpacity += (1.0 - accOpacity) * smpOpacity;
        dist += nextStepLength;
        smpPos += dir * nextStepLength;
        stepLength = nextStepLength;
    }
    return accOpacity;
}
```

Listing 10: Cone tracing algorithm.

Since we're packing all six faces into one texture, the grid's dimensions are not the texture's dimensions but can be easily derived from it:

```
float3 GridDimensions()
{
    float3 gridDim;
    opacityMap.GetDimensions(gridDim.x, gridDim.y, gridDim.z);
    // we index the 6 faces with the X axis coordinate
    gridDim.x /= 6.0;
    return gridDim;
}
```

Listing 11: Computing grid dimensions for a given texture.

To avoid self-intersection, the first sampling location must end up in a different mip 0 voxel than the trace's starting position. In the worst-case scenario, the distance to travel would be the length of a mip 0 voxel's diagonal: $\sqrt{x_v^2 + y_v^2 + z_v^2}$.

However, the minimum distance required might be extremely small when the starting location is adjacent to the next voxel. We don't compute the minimum distance by clipping against the voxel grid but compute the length of mip 0 voxel along the given direction, which ensures that we always move forward by exactly one voxel:

```
float VoxelSizeMip0(float3 gridDim, float3 dir)
{
    // epsilon is a very small positive number used to avoid a division by 0
    float3 voxelDim = 1.0 / gridDim;
    float3 voxelDimOverDir = voxelDim / max(abs(dir), epsilon));
    return min(voxelDimOverDir.x, min(voxelDimOverDir.y, voxelDimOverDir.z));
}
```

Listing 12: Computing the length of a mip 0 voxel along the given direction.

The self-intersection correction logic successfully helps us avoid intersecting against the solid location we start the trace from. However, if another solid surface were to be inside the same starting voxel, we could incorrectly skip past it as illustrated in Figure 5.22.

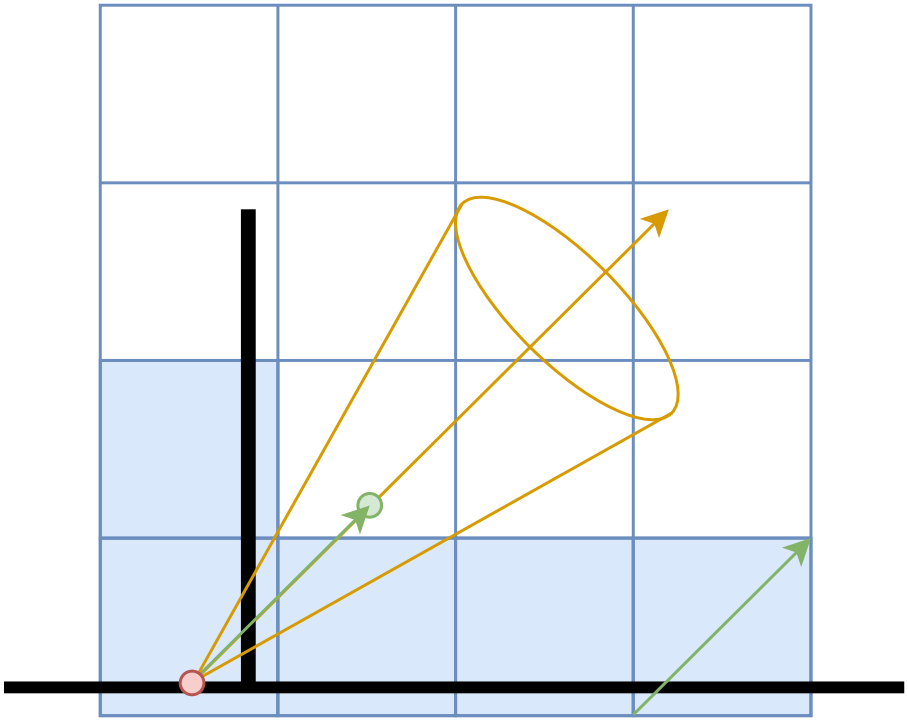
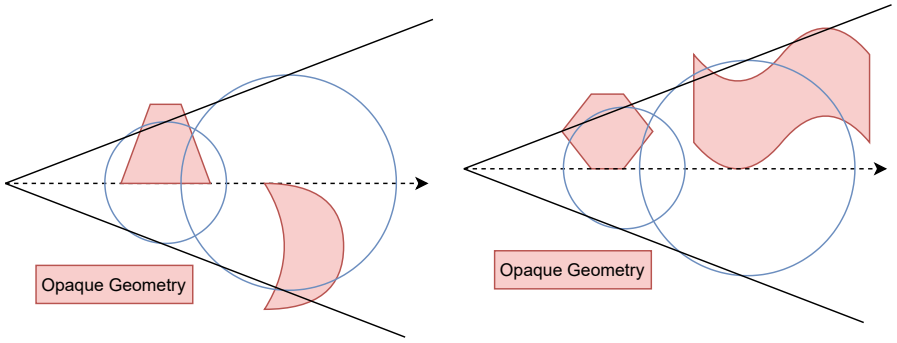


Figure 5.22: Self-intersection avoidance can lead to light leaks. The trace starting from the red point is shifted to the green point, thereby skipping past the vertical geometry and allowing light contributions from behind it.

Since the voxel cone tracing algorithm is an approximation, some extreme cases can lead to heavy amounts of light leaking or over darkening. This happens because opacity is averaged over a surface or volume, which means that we can't properly account for all occlusion scenarios. Figure 5.23 shows the two most severe cases.



(a) Light leak. Opacity should be 100%. **(b)** Over darkening. Opacity should be 50%.

Figure 5.23: In both cases, the final opacity is 75%.

In our implementation, the tracing function operates in a normalized space where all valid locations lie in the $[0, 1]$ range:

```
bool IsPointInScene(float3 p)
{
    const float3 min = float3(0, 0, 0);
    const float3 max = float3(1, 1, 1);
    return all(p >= min) && all(p <= max);
}
```

Listing 13: A point is in the scene if all of its coordinates are in the $[0, 1]$ range.

Reading the voxel data is the most critical and expensive part of the trace. As illustrated in Figure 5.21, we interpolate between two data look-ups from the two closest mip levels in much the same way as what hardware mip-mapping does. See Listing 14. We don't use hardware mip-mapping because our use case is not directly supported. If the LOD ends up being a whole number, the surface of the area to be evaluated matches a mip level exactly and only one

mip needs to be sampled:

```
float SampleOpacity(float lod, float3 tc, float3 dir)
{
    float lowMip = floor(lod);
    float lowValue = SampleOpacityOneMip(lowMip, tc, dir);
    float result = lowValue;
    if(lowMip != lod)
    {
        float highMip = ceil(lod);
        float highValue = SampleOpacityOneMip(highMip, tc, dir);
        result = lerp(lowValue, highValue, frac(lod));
    }
    return result;
}
```

Listing 14: Opacity sampling works by interpolating between the two closest mip values.

When evaluating the data for one mip level, we always end up reading the attributes for three faces. For any given trace direction, there are always three faces whose normal are oriented towards the trace's origin point, one for each axis. We must therefore figure out which three faces to read from and what their contribution weights should be. The trace's direction vector already has three values that represent the importance of each selected face but not in a form directly usable. To apply trilinear filtering on the data from the three faces, we need the sum of all three weights to be equal to 1: $x_w + y_w + z_w = 1$. Since the direction vector is unit length, the following holds true: $\sqrt{x_d^2 + y_d^2 + z_d^2} = 1$. Therefore, $x_d^2 + y_d^2 + z_d^2 = 1$ holds true as well. Trivially, we can set $x_w = x_d^2$, $y_w = y_d^2$, $z_w = z_d^2$. Thus, component-wise multiplication of the direction vector by itself yields the weight vector: $w = d \circ d$. For smooth results, the texture sampler uses trilinear filtering so that each face's attribute is interpolated based

on position, not just direction:

```
float SampleOpacityOneMip(float lod, float3 tc, float3 dir)
{
    float3 weights = dir * dir;
    float3 faceOffsets = dir >= 0.0 ? 0.0 : 1.0;
    float3 tcOffsets = (float3(0.0, 2.0, 4.0) + faceOffsets) / 6.0;
    float3 tcOffsetX = float3(tcOffsets.x, 0.0, 0.0);
    float3 tcOffsetY = float3(tcOffsets.y, 0.0, 0.0);
    float3 tcOffsetZ = float3(tcOffsets.z, 0.0, 0.0);
    float3 texData;
    texData.x = opacityMap.SampleLevel(opacitySampler, tc + tcOffsetX, lod).r;
    texData.y = opacityMap.SampleLevel(opacitySampler, tc + tcOffsetY, lod).r;
    texData.z = opacityMap.SampleLevel(opacitySampler, tc + tcOffsetZ, lod).r;
    return dot(texData, weights);
}
```

Listing 15: Sampling opacity for a single mip level involves reading opacity from the three faces facing the trace's origin using trilinear filtering to interpolate based on position and interpolating the results based on the trace direction.

As alluded to in Section 5.3.6.7, voxel cone tracing is analogous to ray-marching through a volume. Without any correction factor, decreasing the step size leads to an increased accumulation of opacity and emittance. To account for the step size, we use the opacity correction method presented in [Eng+04]:

```
float CorrectOpacity(float opacity, float factor)
{
    return 1.0 - pow(1.0 - saturate(opacity), factor);
}
```

Listing 16: The factor variable is the ratio between the current sampling distance and the normal sampling rate.

5.3.6.9 Indirect Lighting

Our implementation of indirect lighting is split in three stages, one of which is optional. The first stage captures direct lighting into a data structure that can be easily traversed: the dense voxel grid. The last stage does a final gather step by tracing cones into the aforementioned data structure in the deferred shading pass. Optionally, a light bounce can be computed by tracing a voxel grid and writing back to a voxel grid. This is needed when we want to have two or more bounces of indirect lighting, the first bounce being provided by the trace from the deferred pass.

5.3.6.10 Computing More Than One Bounce

When launching a final gather step from the deferred pass into a direct lighting texture, we get one bounce of indirect lighting. In order to compute additional bounces, we can start additional final gather steps either during the deferred pass, or during a computer shader pass following voxelization. If we do so during deferred, we start a new final gather pass every time a cone trace intersects geometry. This leads to a combinatorial explosion of the number of traces: for n cones over the hemisphere and b bounces, the total number of traces is $O(n^b)$. Instead we opted for the compute pass approach, where we launch a final gather for every voxel face. For one bounce in a voxel grid of resolution X, Y, Z and cone count n, the final trace count is at most $6XYZn$ because empty faces can't accumulate light. The number of traces is additive with the number of bounces since a bounce is a single compute dispatch with a fixed workload. The final maximum number of traces accounting for the number of bounces, b, is $6XYZnb$. In order to trace the cones in the right direction, we need the averaged surface normal for every voxel face, hence our normal voxel attribute.

5.3.6.11 Extra Bounces

The bounce pass is a compute shader pre-pass on the emittance map that computes a light bounce in texel-space before the final light bounce happens in the deferred pass. Running this pass once, we get two bounces of light. However, if we want three or more bounces of light, running this expensive pass more than once is problematic. Instead, we run at most a single bounce pass per frame by accumulating bounces until the desired number of bounces is reached. For this to work with any number of bounces, three buffers are needed and we must draw from an older complete result, meaning that the more bounces we have, the more latency indirect lighting will update with. See Figure 5.24 for example schemes of how we handle multi-bounce lighting at a fixed cost.

	Frame 0	Frame 1	Frame 2	Frame 3	Frame 4	Frame 5	Frame 6	Frame 7
Scene	V			V			V	
Buffer 0		B			B			B
Buffer 1			B			B		
Buffer 2								
Buffer Draw Index	0	1	2	2	2	2	2	2
Final Bounce Count	1	2	3	3	3	3	3	3
Frame Delay	0	1	2	3	4	2	3	4

(a) Scheme for three bounce lighting

	Frame 0	Frame 1	Frame 2	Frame 3	Frame 4	Frame 5	Frame 6	Frame 7	Frame 8	Frame 9
Scene	V				V				V	
Buffer 0		B	B			B	B			B
Buffer 1				B				B		
Buffer 2										
Buffer Draw Index	0	1	0	2	2	2	2	2	2	2
Final Bounce Count	1	2	3	4	4	4	4	4	4	4
Frame Delay	0	1	2	3	4	5	6	3	4	5

(b) Scheme for four bounce lighting.

Figure 5.24: Schemes for 3 and 4 bounces of light. **V** is voxelization, **B** is a light bounce that is added to the emittance map. Buffer Draw Index is the buffer that is bound to the deferred pass and used for shading.

5.3.6.12 Ambient Occlusion

Our ambient occlusion [Lan02] implementation traces cones into the opacity map over the hemisphere defined by the local fragment's position and normal. During a trace, each opacity contribution is weighted by the total distance traveled [ZIK98]. Cone trace results are cosine-weighted and summed together. Traces end early when the accumulated occlusion has reached a certain threshold. Our indirect diffuse trace also computes ambient occlusion since the emittance map contains opacity as well. We don't trace the opacity map when indirect diffuse is enabled as it is redundant.

5.3.6.13 Voxel Visualization

When visualizing voxels, we render each opaque face as a rectangle made up of two triangles. Instead of generating data on the CPU, we render a point list with no vertex attributes. We use the vertex id to compute each voxel's 3D grid coordinates in the vertex shader, as seen in Listing 17. We read the voxel texture in the geometry shader for each face and only generate geometry for the non-empty ones. This way, we generate less geometry, and do less texture reads than if we sampled the textures from the pixel shader.

```
uint3 vs_main(uint id : SV_VertexID)
{
    uint numVoxelsInSlice = gridSize.x * gridSize.y;
    uint sliceIndex = id % numVoxelsInSlice;
    uint3 voxelID;
    voxelID.x = sliceIndex % gridSize.x;
    voxelID.y = sliceIndex / gridSize.x;
    voxelID.z = id / numVoxelsInSlice;
    return voxelID;
}
```

Listing 17: Voxel visualization vertex shader, mapping from a vertex id to a voxel id.

5.4 System

The system module is responsible for providing abstractions of system calls to the rest of the application, registering and forwarding input, setting up and registering changes to the window, initializing the rendering module and the scene module, and handling shutdown.

At application start-up, two large chunks of memory are allocated, backed by linear allocators, and split into persistent and transient memory regions. The scene module can then allocate its own data from these two regions. Typically, data that is needed for multiple frames will be allocated in the persistent region, and data that is only needed during a single frame will be allocated from the transient region.

For input, we use the Windows raw input API [Mic20a] by processing the unbuffered read message WM_INPUT [Mic20b] for both mouse and keyboard. In addition, standard input for mouse and keyboard is forwarded to the Dear ImGui library for its own input processing.

Chapter 6

Acceptance Test

The following tables show the result of the application's acceptance test. Most tests for functional and non-functional requirements passed. However, some are not testable as they are too complex to test for the scope of the thesis. These tests were related to the non-functional reliability requirements, which required testing the mean-time-between-failure and mean-time-to-repair. Other tests required testing across diverse systems with mixed hardware specifications, which wasn't possible to do. Note that even though we couldn't test these requirements, they are still part of the application's implementation.

No.	Action	Expected Result	Result	Passed
1	Enable 'Only Ambient Occlusion' from the GUI.	Scene is only drawn with AO.	The scene is drawn with AO.	Passed.
2	Enable either PCSS or PCF filtering and go to an occluded surface.	Shadows are drawn in the occluded area.	Shadows are drawn behind occluders.	Passed.
3	Enable 'Indirect Diffuse' and 'Indirect Specular'.	Scene is illuminated by indirect light.	The scene is illuminated by indirect light.	Passed.
4	Enable 'Use normal map.'	Bumps and dents in surfaces are added to the scene.	Bumps and dents are drawn in surfaces.	Passed.
5	The developer will show that there are no dependencies.	N/A	N/A	Not testable.

No.	Action	Expected Result	Result	Passed
6	The developer will show that the data was pre-processed.	File sizes are optimized.	File sizes have been compressed and the scene loads faster.	Passed.
7	Enable 'PCSS' shadows and inspect the shadows.	Shadows are contact-hardening.	Contact-hardening shadows were drawn	Passed.
8	Enable 'Indirect Diffuse' and 'Indirect Specular'. Change the slider that dictates the number of light bounces.	If the slider is increased, the scene becomes brighter.	The amount of light in the scene changes in relation to the number of bounces.	Passed.
9	Switch between HDR modes.	Performance and quality change depending on the HDR mode.	Emittance results changes in relation to the quality mode.	Passed.

Table 6.3: Acceptance test table for functional requirements.

No.	NFR	Action	Expected Result	Result	Passed
1	U1	Assuming that the GUI is enabled. Change settings.	The drawing of the scene changes in correspondence to the changed setting.	The scene changes in response to change of settings.	Passed.
2	U2	Assuming that the GUI is enabled, see that it is drawn with Dear ImGui.	The GUI is drawn using Dear ImGui.	GUI uses Dear ImGui	Passed.
3	U3	Inspect the GUI.	It takes less than five minutes to understand the GUI.	It took less than five minutes to understand the GUI.	Passed.
4	U4	Use WASD and the mouse to move around the scene.	The camera moves around in response to user input.	The camera moved as it should.	Passed.

No.	NFR	Action	Expected Result	Result	Passed
5	U5	Enable debug visualization modes through the GUI.	Debug visualization modes are drawn.	There are debug visualization modes.	Passed.
6	U6	Inspect the GUI and locate performance and memory usage statistics.	It's possible to assess performance and memory usage.	It was possible to asses performance and memory usage.	Passed.
7	U7	Inspect the GUI and locate light settings. Change the settings.	Lights are updated and the look of the scene changes.	The scenes changes in reponse to light setting changes.	Passed.
8	R1	N/A	N/A	N/A	Not testable
9	R2	N/A	N/A	N/A	Not testable

No.	NFR	Action	Expected Result	Result	Passed
10	P1	Inspect the GUI and locate the frame-time.	FPS should be at an average of 60 or higher.	The average FPS was higher than 60.	Passed.
11	P2	Measure the time it takes to start the application.	It should take less than two seconds to load the scene.	It took less than two seconds to load the scene.	Passed.
12	P3	Use WASD and the mouse to move around the scene.	The response seems to input feels instantaneous.	The response was instantaneous.	Passed.
113	S1	Load the Application on Windows 7/8/10	The application loads.	N/A	Not testable
14	S2	Run the application on multiple devices that support Direct3D feature level 11.0	The application starts and runs fine.	N/A	Not testable.

No.	NFR	Action	Expected Result	Result	Passed
15	S3	The developer explains the architecture and expandability of the application.	The reasoning given by the developer is reliable.	The reasoning is good.	Passed.
16	S4	The developer explains how the GUI works and that it is easily expandable.	The reasoning given by the developer is reliable.	The reasoning is good.	Passed.
17	S5	Run the application on hardware that doesn't support conservative rasterization. Enable software conservative rasterization.	A software implementation of conservative rasterization is used during voxelization.	Software conservative rasterization is there as a substitute to hardware conservative rasterization.	Passed.

Chapter 7

Results and Discussion

7.1 Voxel Cone Tracing

To analyze the performance of the different passes, we get GPU timings for each rendering stage by using hardware timer queries. The main point of interest is voxel cone tracing. All of the operations happening in the bounce pass are for voxel cone tracing, making it trivial to assess the performance. However, the final voxel cone trace in the deferred stage happens as part of the general shading. Therefore, we measure the total GPU time with and without indirect lighting to obtain the time it takes to do the final gather by calculating the difference. We compare performance for all rasterization and quality modes with and without indirect lighting, yielding 18 combinations in total. We collect all GPU timings at an exact location in the scene, without v-sync, and with the same general settings. Furthermore, we keep the 32 most recent timer readings and calculate the median value. The following system setup was used to render the results:

Attribute	Setup
Operating System	Microsoft Windows 10 Home (10.0.18363)
CPU	AMD Ryzen 3950X
GPU	NVIDIA GeForce RTX 2080 Super
GPU Driver Version	497.09
RAM	32 GB
VRAM	8192 MB GDDR 6
Rendering API	Direct3D 11.3

Table 7.2: System setup used for capturing the performance of render stages.

Setting	Value
Screen Resolution	1280x720
Voxel Grid Resolution	128x64x64
Normal Mapping	On
Anisotropic Texture Filtering	4x
Shadow Filtering Mode	PCSS
Number of Bounces	3
Specular Cone Angle	10
Opacity Threshold	0.950
Trace Step Scale	0.5
Mip Bias	0
Deferred Pass Cone Count	9
Bounce Pass Cone Count	9

Table 7.4: Performance impacting settings that don't change in the measurements captured. The last two rows aren't exposed to the UI.

	Time measured for combination (ms)					
	Indirect Lighting (On)			Indirect Lighting (Off)		
	LDR	CHDR	HDR	LDR	CHDR	HDR
Emittance Voxelization	0.41	0.40	8.21			
Emittance Format Fix	0.00	0.10	0.11			
Emittance Opacity Fix	0.06	0.00	0.00			
Emittance Bounce	2.66	5.81	5.68			
Emittance Pre-filtering	0.06	0.11	0.11			
Geometry Pass	0.22	0.22	0.20	0.22	0.22	0.22
Deferred Pass	2.36	2.57	2.56	0.46	0.46	0.46
Full Frame - Voxelization	3.93	4.21	11.72			
Full Frame - Bounce	6.17	9.26	9.13			
Voxel Cone Tracing	1.90	2.11	2.10			

Table 7.6: Measurements for voxel cone tracing related stages with no conservative rasterization enabled. LDR is low dynamic range. CHDR is our custom high dynamic range. HDR is high dynamic range.

	Time measured for combination (ms)					
	Indirect Lighting (On)			Indirect Lighting (Off)		
	LDR	CHDR	HDR	LDR	CHDR	HDR
Emittance Voxelization	0.88	0.63	13.13			
Emittance Format Fix	0.00	0.10	0.11			
Emittance Opacity Fix	0.06	0.00	0.00			
Emittance Bounce	3.33	7.39	7.30			
Emittance Pre-filtering	0.06	0.11	0.11			
Geometry Pass	0.22	0.22	0.19	0.22	0.22	0.22
Deferred Pass	2.09	2.28	2.27	0.55	0.46	0.46
Full Frame - Voxelization	4.14	4.15	16.35			
Full Frame - Bounce	6.56	10.55	10.42			
Voxel Cone Tracing	1.54	1.82	1.81			

Table 7.8: Measurements for voxel cone tracing related stages with software conservative rasterization enabled.

	Time measured for combination (ms)					
	Indirect Lighting (On)			Indirect Lighting (Off)		
	LDR	CHDR	HDR	LDR	CHDR	HDR
Emittance Voxelization	0.81	0.58	13.26			
Emittance Format Fix	0.00	0.10	0.11			
Emittance Opacity Fix	0.06	0.00	0.00			
Emittance Bounce	3.35	7.35	7.23			
Emittance Pre-filtering	0.06	0.11	0.11			
Geometry Pass	0.22	0.22	0.20	0.22	0.22	0.22
Deferred Pass	2.11	2.29	2.27	0.50	0.46	0.49
Full Frame - Voxelization	4.06	4.09	16.49			
Full Frame - Bounce	6.59	10.50	10.41			
Voxel Cone Tracing	1.61	1.83	1.78			

Table 7.10: Measurements for voxel cone tracing related stages with hardware conservative rasterization enabled.

In our implementation, the bounce and voxelization passes never occur in the same frame. They are mutually exclusive, as illustrated in Figure 5.24. To get filtered data for full bounce frame times, we ran a special version of the code that voxelizes the scene once and then only ever runs bounce passes, such that the median value doesn't include a voxelization frame, which would skew the

results. For the voxelization only frames, we simply used the 1 bounce setting.

The tables show that with conservative rasterization disabled, the bounce passes are less computationally expensive. The reason is that when conservative rasterization is enabled, more voxel faces end up being written to, which means that more cones need to be traced in the bounce pass. Since the software and hardware conservative rasterization modes yield sufficiently similar results, the performance profile of the bounce pass for these two modes is very close.

Once tracing starts, the CHDR and HDR modes access the same amount of data, which is why the performance cost for voxel cone tracing is roughly the same. Tracing for LDR is much faster as it stores two times less data than the other quality modes. A visualization of the memory usage can be seen in Figure 5.18. HDR is the worst-performing of all three quality modes in the emittance voxelization stage. This behavior isn't surprising as there is no maximum iteration count for our moving average computation, as discussed in Section 5.3.6.5.

As mentioned in Section 5.2.1, we have a lower level of detail version of our 3D scene, which we use when running the voxelization stages. This solution was exceptional at improving voxelization performance. By reducing our scene's triangle count from 227327 to 57012, we managed to increase the performance of the emittance voxelization stage from 62.57ms to 12.85ms (using software conservative rasterization, and hdr). Table 7.12 shows the performance gain for all conservative rasterization modes and quality modes.

	Time measured for each rasterization mode (ms)					
	Low Res Model			High Res Model		
	LDR	CHDR	HDR	LDR	CHDR	HDR
Off	0.40	0.33	8.23	2.32	1.55	41.78
Software	0.88	0.47	12.85	7.76	2.36	62.57
Hardware	0.80	0.42	13.30	7.38	2.12	92.56

Table 7.12: Measurements for emittance voxelization with and without using the low level of detail version of the scene.

Chapter 8

Conclusion

We presented a method for achieving real-time global illumination called voxel cone tracing. The implementation has been explained in detail, and its components have been analyzed and compared to other approaches. The advantages of voxel cone tracing over other methods like ray-tracing are that the results are noise-free, the performance is quite good, and the approach is very scalable. Tunable parameters include the voxel grid resolution, the number of cascades, the number of cones traced for AO and indirect lighting, the trace's step scale, opacity threshold, mip bias, and maximum mip value, screen-space sub-sampling for diffuse, specular, and AO, grid fraction processed in a frame for the bounce pass, etc. By only doing a final gather in the deferred shading pass and computing bounces in a compute shader, we improved performance for multiple bounce lighting as discussed in Section 5.3.6.11. Much optimization work remains to be done on this compute shader, as it proved to be the one of the biggest bottleneck[1]. We consider the underlying data structure used in our implementation inferior to the cascade method used by McLaren [McL15]. In contrast to the cascade, our dense grid is simpler to implement but lacks accuracy around the camera. Because of this, it would be better to experiment with other data structures for storing voxels, such as sparse voxel octrees [Oli08; Cra+11] or dense grids with clip-map [TMJ98] style toroidal addressing on the voxel textures [Cur14].

[1] As put forth in the results and discussion Chapter 7

8.1 Future Work

Many different features, optimizations, and experiments have been left out for future work. Some of these are very time-consuming, requiring more than a day's work, for little gain in image quality or performance, and some would be great to have but not necessary. This thesis has mainly focused on voxel cone tracing for GI, prioritizing this approach over others, often choosing the most simplistic route. In the future, we would like to experiment with different approaches to voxel cone tracing and study other techniques for achieving global illumination. The following is a list of some of the ideas that we would have liked to try during the development of the application.

8.1.1 Alpha-tested Surfaces and Transparency

To achieve transparency, we could write the actual opacity value when voxelizing instead of writing 1.0 or 0.0. For our 32-bit unorm emittance case, this would require writing to separate textures for emittance and opacity because the fourth channel of the emittance texture is reserved for the fragment counter. To obtain alpha-tested surfaces, we could write a constant value per object instead of 1.0 when writing to an opacity map only. This simplification for alpha-tested surfaces is a common trick for controlling an average opacity level without having to read a texture in the voxelization pixel shader and super-sampling results to get an accurate averaged opacity. This helps the visual quality of ambient occlusion.

8.1.2 True indirect specular

As mentioned in Section 5.3.1 indirect diffuse contributes to a local fragment's specular response. By using another data structure e.g. reflection probes [MMG06], we could achieve true indirect specular.

8.1.3 Amortized Bounce Passes

The computation of a bounce pass can be amortized over multiple frames by tracing over a different sub-section of the voxel volume every frame. If we divide the voxel grid in N sub grids, a full bounce pass takes N frames to complete. While the frame time improves, we increase the time it takes to update indirect lighting.

8.1.4 Emissive Surfaces

With our scheme, emissive surfaces would be drawn as any other, except that no visibility function is needed. In other words, the shading is the same for direct lighting and voxelization but no shadow maps are used.

8.1.5 Short/long Cone Split in the Bounce Pass

As described by McLaren [McL15], for every group of two-by-two, three-by-three, four-by-four, etc. faces, we would trace a long cone from the middle point of the group of faces. For every face, we trace a short cone from its center point and use the long cone's second half to complete the result of the trace. That way, important occluders near the faces aren't missed even though we sub-sample the cone traces.

8.1.6 Pre-weighted Anisotropic Voxels

Another optimization described by McLaren [McL15] is to create a pre-computed voxel grid for every possible sampling direction. Instead of rotating a pre-built hemisphere to trace diffuse cones, we have pre-built cones that cover the entire sphere. When we trace over the hemisphere for diffuse or AO, we test each cone's direction against the normal of the surface. If the dot product is negative, then it's shooting outside the hemisphere and it is rejected. For example, if we have 16 cones over the sphere, then we would use 8 cones for diffuse and AO and we would have 16 pre-weighted voxel grids. When tracing a pre-weighted grid, we only need a single texture fetch per mip level instead of three. In other words, our manual trilinear filtering is pre-computed.

8.1.7 Reading From A Single Mip Level by Choosing the Right Aperture

By picking the cone count and the angle of the cone such that every new sphere's radius is exactly twice the previous one and starting with the first sphere having exactly the diameter of a voxel, we eliminate the need to sample from two mip levels. Since the first sphere has the diameter of a voxel, it lands exactly at mip level 0. Because each new sphere's size is double the previous one, they land exactly at mip levels 1, 2, etc. This scheme halves the texture fetches, but it forces a specific cone count and aperture angle.

8.1.8 Screen-space Sub-sampling

Instead of tracing from the deferred pass, we would trace from a new pass that happens before and write to a lower resolution texture. We would then up-sample the texture using a joint bi-lateral filter [Par+07] that uses the depth and normal vectors to weight the final fragment value. The idea is that the larger the depth difference, the less important the fragment's contribution is. The same idea is applicable to surface normals with angle differences. If the depth or surface angle are too different, then the fragments most probably don't belong to the same object's surface.

8.1.9 Combining with Screen-space Techniques

By combining screen-space cone tracing, which treats the depth buffer as a height map, with voxel cone tracing, it is possible to improve image quality and performance. To improve performance, one could start a screen-space trace for reflections and only run a voxel cone trace when the screen-space trace hasn't found any data in the G-buffer. To improve image quality, a screen-space cone trace, the length of a mip 0 voxel, can be used before the voxel cone trace to prevent the light leaking problems due to self-intersection avoidance. In addition, screen-space diffuse and specular occlusion factors can be computed to correct the final result, as described in [ST15]. Note that indirect lighting in screen-space requires getting color data that hasn't been computed yet for the current frame, therefore a re-projection of the previous frame's data to this frame using motion vectors is required.

8.1.10 Relying On TAA

It would be possible to lean upon TAA to deal with the voxelization process' temporal instability. The maximum number of iterations allowed on the atomic operations we use to average attribute values during the voxelization pass can be reduced since TAA will mitigate the increased flickering that would ensue.

8.1.11 Solid Voxelization

Finally, we would like to experiment with a compute pass to attempt to fill in the gaps inside objects between the voxelization and mip-generation passes [McL15], thereby attempting to get results similar to solid voxelization in order to reduce light leaks. Arbitrary geometry isn't guaranteed be closed manifold, which is why solid voxelization wasn't an option.

Bibliography

[Ama84] John Amanatides. 'Ray tracing with cones'. In: *ACM SIGGRAPH Computer Graphics* 18.3 (1984), pp. 129–135. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.582&rep=rep1&type=pdf.

[Bak16] Dan Baker. 'Object Space Lighting'. In: GDC (April 2016).

[BH13] Christopher A. Burns and Warren A. Hunt. 'The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading'. In: *Journal of Computer Graphics Techniques (JCGT)* 2.2 (August 2013), pp. 55–69. ISSN: 2331-7418. URL: http://jcgt.org/published/0002/02/04/.

[Cal03] Dean Calver. *Photo-realistic Deferred Lighting*. July 2003. URL: https://www.beyond3d.com/content/articles/19/1 (visited on 14th December 2021).

[CCC87] Robert L Cook, Loren Carpenter and Edwin Catmull. 'The Reyes image rendering architecture'. In: *ACM SIGGRAPH Computer Graphics* 21.4 (1987), pp. 95–102.

[Cig+14] Zina H. Cigolle, Sam Donow, Daniel Evangelakos, Michael Mara, Morgan McGuire and Quirin Meyer. 'A Survey of Efficient Representations for Independent Unit Vectors'. In: Journal of Computer Graphics Techniques Vol. 3, No. 2 (2014).

[Cor] Omar Cornut. *Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies*. URL: https://github.com/ocornut/imgui (visited on 14th December 2021).

[Cra+11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green and Elmar Eisemann. 'Interactive Indirect Illumination Using Voxel Cone Tracing'. In: (2011).

[Cur14] Practical Real-Time Voxel-Based Global Illumination for Current GPUs. 'Alexey Panteleev'. In: GTC (2014). URL: https://on-demand.gputechconf.com/gtc/2014/presentations/S4552-rt-voxel-based-global-illumination-gpus.pdf (visited on 14th December 2021).

[Dee+88] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy and Neil Hunt. 'The triangle processor and normal vector shader: a VLSI system for high performance graphics'. In: *Acm siggraph computer graphics* 22.4 (1988), pp. 21–30.

[DS05] Carsten Dachsbacher and Marc Stamminger. 'Reflective shadow maps'. In: *Proceedings of the 2005 symposium on Interactive 3D graphics and games*. 2005, pp. 203–231.

[Eng+04] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Aaron E. Lefohn, Christof Rezk Salama and Daniel Weiskopf. *Course Notes 28*. SIGGRAPH Real-Time Volume Graphics course. August 2004, p. 30.

[Eng09] Wolfgang Engel. 'Designing a renderer for multiple lights: The light pre-pass renderer'. In: *Shader X7* (2009). Ed. by Wolfgang Engel.

[Fer05] Randima Fernando. 'Percentage-closer soft shadows'. In: *ACM SIGGRAPH 2005 Sketches*. 2005, 35–es.

[HAO05] Jon Hasselgren, Tomas Akenine-Möller and Lennart Ohlsson. *Conservative Rasterization. GPU Gems 2*. March 2005. Chap. 42.

[Har96] John C Hart. 'Sphere tracing: A geometric method for the anti-aliased ray tracing of implicit surfaces'. In: *The Visual Computer* 12.10 (1996), pp. 527–545.

[HMY12] Takahiro Harada, Jay McKee and Jason C. Yang. 'Forward+: Bringing Deferred Lighting to the Next Level'. In: *Eurographics 2012 - Short Papers*. Ed. by Carlos Andujar and Enrico Puppo. The Eurographics Association, 2012. DOI: 10.2312/conf/EG2012/short/005-008.

[Hol11] Daniel Holbert. 'Saying 'Goodbye'to Shadow Acne'. In: *Game Developers Conference poster*. 2011, p. 238.

[Jim14] Jorge Jimenez. 'Next Generation Post Processing in Call of Duty: Advanced Warfare'. In: SIGGRAPH Advances in Real-Time Rendering in Games course (August 2014), p. 120.

[Kaj86] James T Kajiya. 'The rendering equation'. In: *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. August 1986, pp. 143–150.

[Kar+] Branimir Karadžić et al. *GENie - Project generator tool*. URL: https://github.com/bkaradzic/genie (visited on 14th December 2021).

[Kar14] Brian Karis. 'High Quality Temporal Supersampling'. In: SIGGRAPH Advances in Real-Time Rendering in Games course (August 2014).

[Kar18] Baldur Karlsson. *RenderDoc*. 2018. URL: https://renderdoc.org/ (visited on 20th November 2021).

[Lam60] Jean-Henri Lambert. *JH Lambert,... Photometria, sive de Mensura et gradibus luminis, colorum et umbrae*. sumptibus viduae E. Klett, 1760.

[Lan02] Hayden Landis. 'Global illumination in production'. In: *ACM SIGGRAPH*. 2002.

[McG16] Morgan McGuire. *McGuire Computer Graphics Archive. Crytek Sponza*. 2016. URL: https://casual-effects.com/g3d/data10/common/model/crytek_sponza/sponza.zip (visited on 27th November 2021).

[McL15] James McLaren. 'The Technology of The Tomorrow Children'. In: GDC (April 2015).

[Mic] Microsoft. *Percentage Closer Filtering*. URL: https://microsoft.github.io/DirectX-Specs/d3d/archive/D3D11_3_FunctionalSpec.htm#22.4.19%20sample_c (visited on 14th December 2021).

[Mic20a] Microsoft. *Raw Input*. August 2020. URL: https://docs.microsoft.com/en-us/windows/win32/inputdev/using-raw-input (visited on 13th December 2021).

[Mic20b] Microsoft. *WM_INPUT message*. August 2020. URL: https://docs.microsoft.com/en-us/windows/win32/inputdev/wm-input (visited on 13th December 2021).

[Mic21a] Microsoft. *D3D12Device::CreateGraphicsPipelineState method*. October 2021. URL: https://docs.microsoft.com/en-us/windows/win32/api/d3d12/nf-d3d12-id3d12device-creategraphicspipelinestate (visited on 13th December 2021).

[Mic21b] Microsoft. *ID3D11Device::CreateQuery method (d3d11.h)*. October 2021. URL: https://docs.microsoft.com/en-us/windows/win32/api/d3d11/nf-d3d11-id3d11device-createquery (visited on 13th December 2021).

[Mit12] Martin Mittring. 'The Technology Behind the "Unreal Engine 4 Elemental demo"'. In: SIGGRAPH Advances in Real-Time Rendering in Games course (August 2012).

[MMG06] Jason Mitchell, Gary McTaggart and Chris Green. 'Shading in valve's source engine'. In: *Acm siggraph 2006 courses*. August 2006, pp. 129–142. URL: https://advances.realtimerendering.com/s2006/Mitchell-ShadingInValvesSourceEngine.pdf.

[Nic65] Fred E. Nicodemus. 'Directional reflectance and emissivity of an opaque surface'. In: *Applied Optics* 4.7 (July 1965), p. 767. DOI: 10.1364/AO.4.000767.

[NVI] NVIDIA. *NVIDIA Texture Tools Exporter*. URL: https://developer.nvidia.com/nvidia-texture-tools-exporter (visited on 14th December 2021).

[Oli08] Jon Olick. 'Current Generation Parallelism In Games'. In: Beyond Programmable Shading: In Action (August 2008). URL: https://www.jonolick.com/uploads/7/9/2/1/7921194/olick-current-and-next-generation-parallelism-in-games.pdf (visited on 14th December 2021).

[Par+07] Sylvain Paris, Pierre Kornprobst, Jack Tumblin and Frédo Durand. 'A gentle introduction to bilateral filtering and its applications'. In: *ACM SIGGRAPH 2007 courses*. 2007, pp. 39–40. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.221.7094&rep=rep1&type=pdf.

[Pet10] Matt Pettineo. *Attack of the depth buffer*. March 2010. URL: https://therealmjp.github.io/posts/attack-of-the-depth-buffer/ (visited on 28th November 2021).

[Pho75] Bui Tuong Phong. 'Illumination for computer generated pictures'. In: *Communications of the ACM* 18.6 (1975), pp. 311–317.

[RSC87] William T Reeves, David H Salesin and Robert L Cook. 'Rendering antialiased shadows with depth maps'. In: *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. 1987, pp. 283–291.

[Sch13] Christian Schüler. 'Followup: Normal Mapping Without Precomputed Tangents'. In: (2013). URL: http://www.thetenthplanet.de/archives/1180.

[Sch94] Christophe Schlick. 'An inexpensive BRDF model for physically-based rendering'. In: *Computer graphics forum*. Vol. 13. 3. Wiley Online Library. 1994, pp. 233–246.

[Sou11] Tiago Sousa. 'Anti-Aliasing Methods in CryENGINE 3'. In: Courses: Filtering Approaches for Real-Time Anti-Aliasing (August 2011).

[ST15] Ari Silvennoinen and Ville Timonen. 'Multi-Scale Global Illumination in Quantum Break'. In: SIGGRAPH Advances in Real-Time Rendering in Games course (August 2015).

[Tak13] Dean Takahashi. *How Pixar made Monsters University, its latest technological marvel*. 2013. URL: https://venturebeat.com/2013/04/24/the-making-of-pixars-latest-technological-marvel-monsters-university/ (visited on 13th October 2021).

[TMJ98] Christopher C Tanner, Christopher J Migdal and Michael T Jones. 'The clipmap: a virtual mipmap'. In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. 1998, pp. 151–158.

[Vul21] Vulkan. *10.2. Graphics Pipelines*. December 2021. URL: https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/vkspec.html#pipelines-graphics (visited on 13th December 2021).

[Whi79] Turner Whitted. 'An improved illumination model for shaded display'. In: *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*. 1979, p. 14.

[Wil78] Lance Williams. 'Casting curved shadows on curved surfaces'. In: *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*. 1978, pp. 270–274. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.134.8225&rep=rep1&type=pdf.

[Yan+09] Lei Yang, Diego Nehab, Pedro V Sander, Pitchaya Sitthi-Amorn, Jason Lawrence and Hugues Hoppe. 'Amortized supersampling'. In: *ACM Transactions on Graphics (TOG)* 28.5 (2009), pp. 1–12.

[ZIK98] S. Zhukov, A. Iones and G. Kronin. 'An ambient light illumination model'. In: *Rendering Techniques '98*. Ed. by G. Drettakis and Nelson Max. Vienna: Springer Vienna, 1998, pp. 45–55. ISBN: 978-3-7091-6453-2.

Acronyms

1D One Dimensional.

2D Two Dimensional.

3D Three Dimensional.

AO Ambient Occlusion.

API Application Programming Interface.

AR Augmented Reality.

BPP Bits Per Pixel.

BRDF Bidirecitonal Reflectance Distribution Function.

CAS Compare And Swap.

CAD Computer Aided Design.

CGI Computer Generated Imagery.

CHDR Compressed High Dynamic Range.

CPU Central Processing Unit.

DDS DirectDraw Surface.

FIFO First In First Out.

FPS Frame Per Second.

GBUFFER Geometry Buffer.

GI Global Illumination.

GPU Graphical Processing Unit.

GUI Graphical User Interface.

HDR High Dynamic Range.

HLSL High-Level Shader Language.

IGN Interleaved Gradient Noise.

LDR Low Dynamic Range.

LOD Level Of Detail.

MB Megabyte.

MSAA Multisample Anti-Aliasing.

MS Milliseconds.

MTBF Mean Time Between Failure.

NVTTE NVIDIA Texture Tools Exporter.

OBJ Wavefront Geometry File Format.

OOP Object Oriented Programming.

OS Operating System.

PCF Percentage Closer Filtering.

PCSS Percentage Closer Soft Shadows.

PSO Pipeline State Object.

RAM Random Access Memory.

RGBA Red Green Blue Alpha.

RTV Render Target View.

SRV Shader Resource View.

SVO Sparse Voxel Octree.

TAA Temporal Anti-Aliasing.

UAV Unordered Access View.

UI User Interface.

UNORM Unsigned Normalized Integer.

VCT Voxel Cone Tracing.

VRAM Video Random Access Memory.

VR Virtual Reality.