TESSELLATED VOXELIZATION FOR GLOBAL ILLUMINATION USING VOXEL CONE TRACING

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Sam Freed

June 2018

COMMITTEE MEMBERSHIP

TITLE:	Tessellated Voxelization for Global Illumination using Voxel Cone Tracing
AUTHOR:	Sam Freed
DATE SUBMITTED:	June 2018
COMMITTEE CHAIR:	Christian Eckhardt, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Maria Pantoja, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Aaron Keen, Ph.D. Professor of Computer Science

ABSTRACT

Tessellated Voxelization for Global Illumination using Voxel Cone Tracing

Sam Freed

Modeling believable lighting is a crucial component of computer graphics applications, including games and modeling programs. Physically accurate lighting is complex and is not currently feasible to compute in real-time situations. Therefore, much research is focused on investigating efficient ways to approximate light behavior within these real-time constraints.

In this thesis, we implement a general purpose algorithm for real-time applications to approximate indirect lighting. Based on voxel cone tracing [13], we use a filtered representation of a scene to efficiently sample ambient light at each point in the scene. We present an approach to scene voxelization using hardware tessellation and compare it with an approach utilizing hardware rasterization. We also investigate possible methods of warped voxelization.

Our contributions include a complete and open-source implementation of voxel cone tracing along with both voxelization algorithms. We find similar performance and quality with both voxelization algorithms.

ACKNOWLEDGMENTS

Thanks to:

- My family, for always supporting me
- My friends, for the encouragement, laughs, and fun
- The wonderful professors at Cal Poly, especially Dr. Zoë Wood and my advisor Dr. Christian Eckhardt
- Andrew Guenther, for uploading this template

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

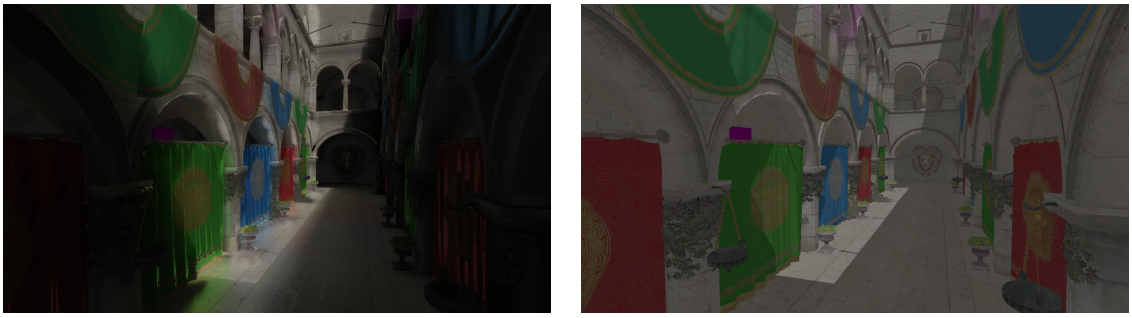
Computer graphics is the task of taking a virtual description of a scene, composed of objects such as models and lights, and rendering that scene to an image. One of the core components involved with this is computing realistic lighting. Although light behavior is fairly well understood in the physical sense, accurately simulating this behavior is a heavy computational task. The goal in computer graphics, then, is to efficiently approximate light behavior where there are time and resources constraints.

Of particular importance in computer graphics are real-time applications, where images, or frames, must be produced at an interactive speed, such as in video games. Typically, the lower bound for this is considered 30 frames per second, or about 33 milliseconds per frame, with a target of 60 frames per second, or about 17 milliseconds per frame. To meet this goal, researchers and engineers have designed many algorithms and techniques. In this work, we implement and extend an algorithm to accomplish fast and accurate lighting for fully dynamic scenes. An example of the result of our global illumination algorithm is shown in Figure 1.1.

It is important to keep in mind that most lighting algorithms have many tradeoffs and are adapted towards specific use cases. A complete lighting system may also combine many different lighting algorithms in order to achieve the desired balance between quality, performance, and ease of use.

1.1 Real-Time Global Illumination

In general, lighting at a given point is a combination of direct and indirect light. Direct light is light accumulated directly from a light source, whereas indirect light is



(a) Global illumination (b) No global illumination

Figure 1.1: A scene—describing information like light sources, materials, and geometry—is rendered by computing a color for each pixel of a screen. The image on the left is rendered with global illumination whereas the image on the right uses a constant ambient term for indirect lighting.

the light that comes from other non-light sources in the scene (e.g. the light that has 'bounced' off of objects in the scene).

In the beginnings of computer graphics simple lighting models were used in order to maintain real-time performance. While the direct light was feasible to compute in real-time (albeit with limitations on number of lights and other optimizations), indirect lighting was often faked with a simple constant contribution. As hardware and algorithms improved, more advanced techniques were developed and approximating the indirect light became more feasible. Two methods, ambient occlusion [7, 5] and baked lighting [43, 5], became standard ways of introducing simple global illumination, the term given to techniques that improved indirect lighting. However, both of these methods have drawbacks. Ambient occlusion, which approximates the effect of light occlusion by nearby objects, provides a great improvement over the constant ambient term used previously but does not accurately simulate the indirect light in its entirety. Thus it is only 'partial' global illumination. Baked lighting involves pre-computing light behavior of a scene prior to runtime and then utilizing the precomputation to enable real-time realistic lighting. The drawback here is only static objects can have their light baked—any dynamic objects or lights in the scene are not accounted for.

Full global illumination algorithms attempt to provide a complete approximation of indirect light. Some, like baked lighting, only work for static scenes. For our work, however, we are interested in dynamic global illumination. In other words, our goal is full global illumination that can be performed completely for each frame. Any changes in the scene—object movement, light movement or changes in intensity—will be accounted for.

For dynamic real-time global illumination, a common approach is to first construct a spatial representation of a scene's radiance (light information) and then use that representation to approximate the radiance at a given point in the scene. Popular methods that follow this approach are light propagation volumes [24] and voxel cone tracing [13]. Some challenges that arise from methods like these are issues with GPU memory consumption and achieving adequate lighting detail.

1.2 Our Contribution

The main contribution of this work is a complete implementation of computing global illumination based on voxel cone tracing with a focus on ease of use and speed while still producing high quality results. We hope this provides a competitive alternative to existing real-time global illumination systems that can be used for educational purposes.

We also present an approach to scene voxelization using hardware tessellation and compare it with an approach utilizing hardware rasterization. We find similar performance to the raster approach but with promising results for perspective warped voxels. We also investigate possible methods of nonuniform voxelization.

Another important part of our contribution is that the implementation is open-source and cross platform (on Linux and Windows using modern OpenGL[1]). Many

[1] macOS is not supported since Apple only supports up to OpenGL 4.1 [1], whereas we require 4.5

graphics implementations are developed in the research or industrial space and are often not made available to the public for various reasons (e.g. copyright). Even implementations that are openly available often are tied to a specific game engine or other large codebase[2], making it difficult to understand or integrate into one's own project. Furthermore, many implementations utilize DirectX as their graphics API, which limits the implementation to Windows only. We hope that providing an easy to understand and cross platform implementation will help others learn more about voxel cone tracing and dynamic real-time global illumination.

(for features like compute shaders, direct state access, and image objects).

[2]For example, NVIDIA's VXGI is integrated into a custom branch of Unreal Engine 4 (distributed as a binary) [2] and Light Propagation Volumes are implemented in CryEngine3 [23]. The Godot engine [20] is a notable exception here and, while part of a large codebase, is relatively small and is completely open source.

Chapter 2

BACKGROUND

2.1 Representing Geometry

2.1.1 Triangles

To develop an interactive 3D application a scene representation is needed. Traditionally, all geometric objects in a scene are represented by triangles. For example, to model a simple cube we can represent each of its faces using two triangles, for a total of 12 triangles. The reason for using triangles is due to their geometric simplicity (triangles contain the fewest number of points that define a plane). Also, as dedicated Graphics Processing Units (GPUs) became more common they were designed with this traditional triangle rasterization in mind and have specialized hardware to operate on triangles. In other words, triangles are fast to process.

But triangles have some issues. Primarily, they do a good job of representing surfaces (since they are inherently 2D) but they don't lend themselves well to volumetric (3D) data. For example, a natural representation of a cloud would be a 3D volume filled with the density of the cloud at a given point within the volume. There are algorithms that can convert from a volumetric representation to a triangle one—marching cubes [30] being the most popular—but it still only models an arbitrary isosurface as opposed to the actual volume.

2.1.2 Voxels

Voxels (volume elements) represent 3D objects in a natural way. A volumetric representation of an object is a 3D grid of cells (the voxels) which hold any data

relevant to that voxel: color, transparency, and normal, to name a few. Recently, voxels have grown popular in the computer graphics field due to this natural representation of 3D objects. The main reasons voxels were not used much in the past were the amount of memory required to store a voxelized representation and GPUs being specialized for triangle rasterization. This restriction has largely been lifted since modern GPUs have much more memory and general purpose GPU (GPGPU) computing has allowed programmers to work more easily with non-triangle based data and algorithms [12, 10].

2.2 Computer Graphics Primer

This thesis has a heavy focus on implementation details and requires a solid foundation in understanding computer graphics. Thus this brief primer will introduce the most important topics including the graphics pipeline, transforms, and some common graphics terminology and techniques. For more information, we recommend Akenine-Möller et al.'s Real-Time Rendering [5] for a complete survey of real-time rendering and the OpenGL and GLSL specifications [42, 27] for reference.

2.2.1 The Graphics Pipeline and Rasterization

The ultimate goal of most graphics work is to produce a 2D image from some set of input data, typically a scene filled with triangle meshes and various light sources. This process is accomplished in several distinct stages which compose the classic graphics pipeline, some of which are configurable through shaders (a shader is a program that runs on the GPU). We give a brief overview of the pipeline here, but encourage reference to [5] for a more detailed explanation. A flowchart of the graphics pipeline is shown in Figure 2.1.

The input to the pipeline is a series of vertices. Each vertex contains associated information such as its position in 3D space, a surface normal, and a texture coordinate.

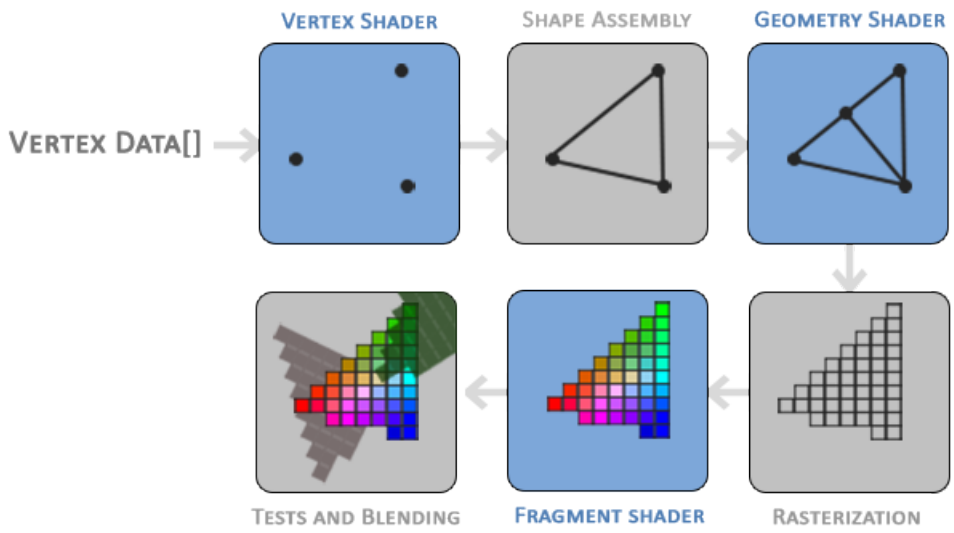


Figure 2.1: Overview of the main stages of the graphics pipeline showing the rasterization of a single triangle (source by Joey de Vries, CC BY 4.0 [15]).

The first step in the pipeline is **vertex shading**, where a vertex shader is executed for each input vertex. Most vertex shaders are fairly simple and perform the job of transforming the input vertex's position from one coordinate system to another.

After vertex shading, **shape assembly** occurs and forms geometric primitives from the vertices. The most common primitive is the triangle, although other options such as points and lines exist.

Optionally, **geometry shading** can take place to operate on the primitives generated from shape assembly. Some common uses here are to introduce more primitives into the pipeline or to modify the input primitive before proceeding to rasterization.

Rasterization involves generating fragments for each primitive that will end up on the screen[1]. Each fragment corresponds to one pixel in the final rendered image and is generated by testing whether a given pixel is contained within an input primitive. The resolution of the fragments is determined by the viewport resolution, which is

[1]The primitives go through clipping immediately before rasterization, which removes fragments outside the screen

generally set to the resolution of the framebuffer being rendered to.

During **fragment shading**, a fragment shader is executed for each fragment generated during rasterization. The output of this shader is typically the color of the pixel for the final image, which is written into a framebuffer. This stage is also where the majority of shading effects occur, including lighting.

Finally, each shaded fragment goes through a **testing and blending** phase. Depth testing is used to determine whether one fragment is in front of or behind the value currently in the framebuffer. Alpha testing and blending occur when the fragment is transparent (the alpha, or opacity, value is less than one). The tests and blending can be configured and enabled/disabled as desired (but is not performed programmatically in a shader).

2.2.2 Transforms

A crucial part of computer graphics is transforming points from one coordinate system to another. Transformations are needed to ensure objects are placed correctly in the virtual world and then to project them from 3D space into a 2D space that can be rasterized and displayed on the screen. Most transformations are represented as matrices. Therefore applying a transformation T to a particular vector $\boldsymbol{v}$ is just multiplication, $T\boldsymbol{v}$. Note that OpenGL uses a right-handed coordinate system: if the coordinate system is aligned with our screen then the x axis points right, y points up, and z points *out of* the screen (the 'forward' direction is $-z$).

Some important coordinate systems for computer graphics and this work, some of which are shown in Figure 2.2 are:

Object Space: the local coordinate system for an object. This is the initial space all vertex coordinates are in when loaded from a mesh.

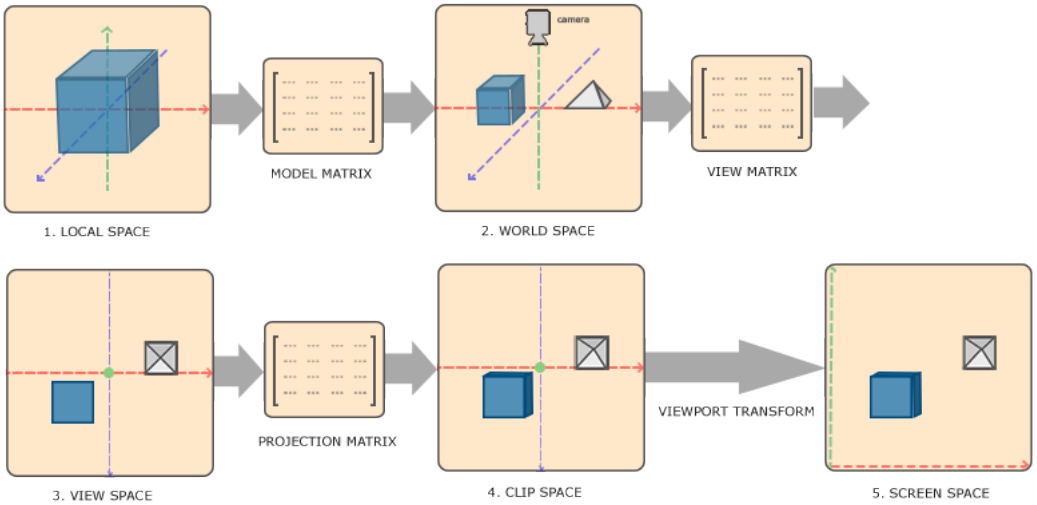


Figure 2.2: Overview of the main coordinate systems used during rasterization and the method of transformation between them (source by Joey de Vries, CC BY 4.0 [15]).

World Space: the global coordinate system for the virtual world. Objects are placed into the world using a **model** matrix.

View Space: the virtual world given from the perspective of a camera (the viewer). In this space, the camera is located at the origin, $(0, 0, 0)$, and faces in the $-z$ direction. The transformation from world space to view space is accomplished using a **view** matrix.

Normalized Device Coordinates (NDC): a coordinate system in the range $[-1, 1]$ that defines the region that will be sent to be rasterized. Also, the z axis is 'flipped' compared to the standard OpenGL right-handed coordinate system. A **projection** matrix is used to transform from view space to clip space[2] to NDC. There are two general kinds of projection matrices: perspective, which emulates how objects further away from a camera appear smaller, and orthographic, which preserves size regardless of distance from camera.

[2]Clip space is the intermediate coordinate space between view space and NDC. A homogeneous divide by the fourth component of a vector is used to transform from clip space to NDC: $(x_c, y_c, z_c, w_c) \rightarrow (\frac{x_c}{w_c}, \frac{y_c}{w_c}, \frac{z_c}{w_c}, 1)$. If any coordinate is outside the range $[-1, 1]$ the coordinate is 'clipped'; the rasterizer will not produce fragments for those clipped coordinates.

Screen Space: the coordinate space representing a typical 2D screen or image. The x and y coordinates are the respective pixel position (treating the bottom left corner as $(0, 0)$). The z axis is usually interpreted as the depth of the pixel as a value within the range $[0, 1]$. The conversion from clip space to screen space is done automatically by the GPU during rasterization based on the viewport resolution.

Texture Space: the coordinate space used when sampling a value from a texture. The coordinates are within the range $[0, 1]$, with $(0, 0)$ being the bottom left corner. Sampling outside this range results in behavior defined by a texture's wrapping mode (e.g. clamp to the edge value or repeat the texture).

Image Space: the coordinate space used when indexing into an image. The range is defined by the target image and uses integral numbers.

Light Space: similar to the view space, but instead of the camera defining the origin and orientation a light source is used.

Tangent Space: a coordinate system defined with respect to a surface and its normal.

2.2.3 Compute Shaders

While most graphics work follows the graphics pipeline, it is possible to use a more abstract model suited towards general purpose computing. Compute shaders provide an interface to performing this GPGPU work (similar to how CUDA exposes GPGPU operation). At the most basic level, dispatching a compute shader launches many threads that all execute the same program, or kernel, defined in a compute shader. Besides the built-in inputs for determining a thread's location within the compute space, all inputs and outputs are accessed through explicitly defined objects such as textures, images, or buffers.

2.2.4 Tessellation

An optional stage in the pipeline between vertex shading and geometry shading is tessellation. The goal of tessellation is to take a patch—a fixed number of vertices—and subdivide the patch into more vertices. The number of vertices generated from the subdivision is controlled either from a fixed constant (defined in the application) or from the optional **tessellation control** shader. The generated vertices are then passed through the **tessellation evaluation** shader, which performs vertex processing just like the vertex shader.

A triangular patch (defined by three vertices) is subdivided into multiple subtriangles. There are four total tessellation levels: one which corresponds to the inner tessellation and three for the outer tessellation. Loosely speaking, the inner tessellation determines how many inner triangles are generated and the outer tessellation determines how many vertices are placed along each outer edge. See Figure 2.3 for an example.

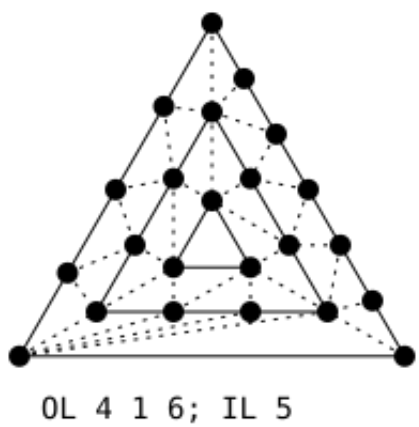


Figure 2.3: A triangle tessellated with an inner tessellation level of 5 and outer tessellation levels of 4, 1, and 6. The dashed lines show the new triangles generated from the original triangle. (image from OpenGL Wiki [3]).

2.2.5 Textures and Mipmapping

A texture in the most basic sense is a linear data buffer. In computer graphics, the most common texture type is a 2D texture where each **texel**—a single element of a texture—stores a color and opacity value, requiring four components—three for the RGB color and one for the opacity. However, 1D and 3D textures are also possible (and 3D textures are a core part of this thesis) as well as various data formats. Textures can also contain multiple **levels**, often used for mipmapping. OpenGL and GPUs have native support for working with multi-level textures and sampling from them appropriately.

Typically, textures can be considered read-only: they are only sampled from inside a shader. To write to a texture, there are two common approaches: render-to-texture and binding the texture as an image. Render-to-texture refers to attaching the texture to a framebuffer object which can then be used as the target for rasterization. The other method involves binding a single level of a texture as an image object which can then be written to inside a shader (usually a fragment or compute shader).

Mipmapping involves keeping multiple detail levels of the same source data. The base level is the highest resolution available and each successive level is downscaled (typically in powers of two: a texture with a base level resolution of 256x256 would have mipmaps with resolutions of 128x128, 64x64, and so on). While this does require increased memory usage, it can smooth out aliasing issues caused by sampling. Mipmapping also generally increases performance as it improves GPU texture cache locality.

2.3 Spatial Data Structures

Many algorithms in computer graphics require a way to query aspects of a particular point or object in space. These queries could be something like "which side of a plane is this object on" or, more relevant to our work, "how much light is present at this point". Of course, this data could be stored in a simple array but oftentimes this leads to poor runtime performance. Using a more advanced data structure to represent spatial data is crucial to achieving real-time rendering performance. We briefly describe some common data structures relevant to global illumination.

2.3.1 3D Textures

One of the simplest approaches to storing spatial data is to simply define a uniform mapping from 3D space into a 3D texture. While this is straightforward to visualize and is easy to work with, 3D textures require a large amount of memory. If the data to be stored is sparse (i.e. most positions in space do not have a corresponding or useful value) this memory is wasted. However, GPUs are able to take advantage of spatial coherency in 3D textures and can provide fast caching and filtering of the data.

2.3.2 Clipmaps

In a 3D texture we essentially have a uniform grid of values: the resolution, or level of detail, of data remains constant throughout the entire texture. With many types of data we don't need the same resolution—for example, in many applications, the farther away from the viewer we are the less detail we need. A clipmap [46, 31] is based on this concept and represents 3D space using multiple levels of detail. The resolution at each level of detail is halved, reducing the amount of memory needed to represent the same 3D space as an equivalent 3D texture. The issue with sparse

memory still exists, but it is alleviated greatly. Figure 2.4 shows a diagram of the multiple levels of detail in a clipmap.

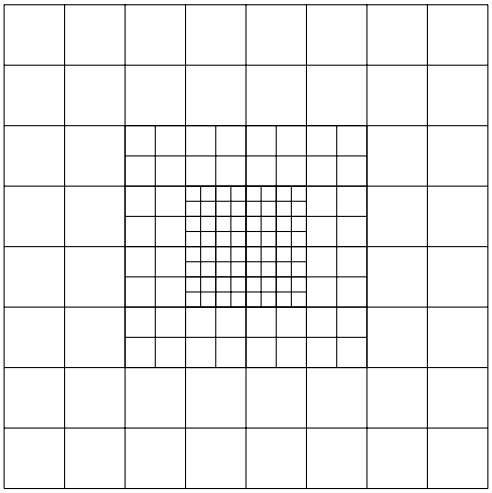


Figure 2.4: A 2D diagram showing the multiple resolutions of a clipmap. Notice that the inner levels have higher detail while each successive outer level lowers the detail level by a factor of 2.

2.3.3 Octrees

Another way of representing 3D space can be done using a tree structure. An octree [34] is a tree where each child node is an octant of its parent (see Figure 2.5). This representation is very efficient memory wise when it comes to sparse data since there is no requirement the tree has to be full: if there is no data in an octant it can be left empty. The tradeoff, however, is the need to recurse through the octree for every query. Both building and recursing an octree can be difficult to do efficiently on the GPU and does not leverage the hardware support for caching and filtering of 3D textures.

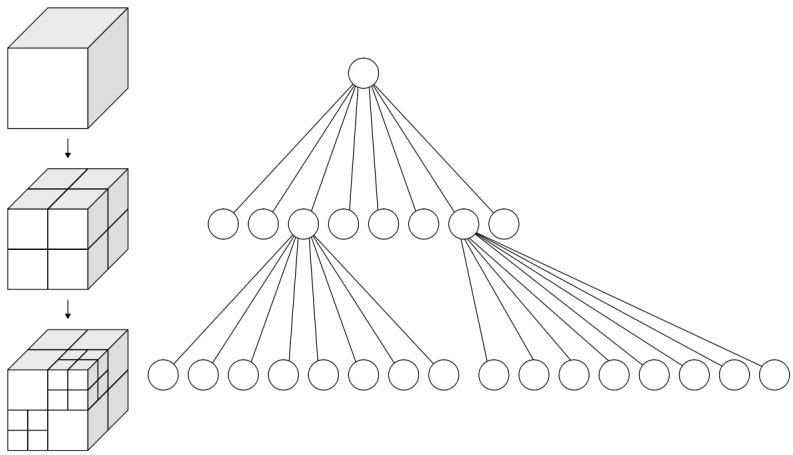


Figure 2.5: Diagram of a simple octree (image distributed under CC BY-SA 3.0 [47]).

2.4 Radiance and the Rendering Equation

As our goal is to accurately model light in our virtual scene, it makes sense to base our shading model on the behavior of real light. The theoretical basis for understanding and measuring light is confined within the field of radiometry. Fortunately, for our purposes, we can focus more on the applied aspects of radiometry for computer graphics—for example, the wave-like nature of light is mostly ignored in favor of particle based behavior.

Fundamentally, light is energy. Therefore, our goal is to compute the amount of light energy at a surface point from a given viewpoint (our camera). This quantity, called radiance, is the instantaneous amount of light energy emitted along the ray of light from the point to the camera. Kajiya [21] presents this problem mathematically in the rendering equation:

$$L_o(\boldsymbol{x}, \omega_o) = L_e(\boldsymbol{x}, \omega_o) + \int_\Omega f_r(\boldsymbol{x}, \omega_i, \omega_o)\ L_i(\boldsymbol{x}, \omega_i)\ (\omega_i \cdot \boldsymbol{n})\ d\omega_i$$

where

$\boldsymbol{x}$ is a point in space,

ω_o is the direction of outgoing light from $\boldsymbol{x}$,

ω_i is the direction of incoming light to $\boldsymbol{x}$,

$\boldsymbol{n}$ is the surface normal at $\boldsymbol{x}$,

L_o is the total radiance at $\boldsymbol{x}$ in the direction ω_o,

L_e is the emitted radiance at $\boldsymbol{x}$ in the direction ω_o,

L_i is the incoming light (irradiance) at $\boldsymbol{x}$ from direction ω_i,

f_r is the bidirectional reflectance distribution function (BRDF) at $\boldsymbol{x}$, and

Ω is the unit hemisphere at $\boldsymbol{x}$ centered around $\boldsymbol{n}$.

Although it looks complex, the result L_o is radiance (our final shaded color) at a given point when viewed from a particular angle (the view vector), the term L_e accounts for emissive light (e.g. from emissive materials), and the integral is calculating how much incoming light at a point is transferred towards the viewer. The dot product $(\omega_i \cdot \boldsymbol{n})$—which is $\cos(\theta_i)$, where θ_i is the angle between ω_i and $\boldsymbol{n}$—attenuates the incoming light based on the angle between the incoming light and the normal (as light direction and normal become perpendicular less light hits the surface).

In order to compute a solution to the rendering equation, the problem must first be discretized. Thus the integral will be approximated by a finite sum over the incoming light. Recall that, in general, the incoming light is broken into two parts: direct light and indirect light. To compute the direct light, the contributions from each light source are summed together: this is accomplished with a straightforward loop over each light. The indirect light, however, is more difficult as there are effectively infinite indirect light sources due to light scattering, reflections, and so on. Naturally, this means there is great interest in being able to accurately and efficiently compute indirect lighting. For real-time applications heavy approximations are made and it

can be difficult to connect the rendering equation with the resulting approximation. To better illustrate the connection, as well as cover a core technique also adapted for use in real-time applications, is helpful to understand raytracing.

2.5 Raytracing

Raytracing is a common alternative to rasterization and is a natural way of calculating light. The idea is for each pixel of our final rendered image we cast a ray from the camera into our scene. If the ray hits an object, the object's material and the light sources in the scene can be used to calculate a color value. Using this model it is simple to extend the basic shading model to support shadows, reflections, and refractions.

Predictably, the downside of raytracing is a high computational cost as calculating ray-object intersections and performing advanced lighting calculations becomes overwhelming. Various techniques to reduce the amount of computation done, such as representing the scene inside a spatial data structure, help tremendously but performance is still not acceptable in real-time applications. Still, many real-time techniques do perform some raytracing in limited capacities: SSR (screen space reflections) and voxel cone tracing being two relevant examples.

2.5.1 Monte Carlo Raytracing

The indirect lighting can be computed in an intuitive way using Monte Carlo raytracing, which approximates the integral from the rendering equation in a straightforward fashion. Ideally, to calculate the integral over the hemisphere Ω, an infinite number of rays would be cast from the point of interest $\boldsymbol{x}$ and the result would be the accumulation of all the ray's values. With Monte Carlo raytracing, the goal is to approximate the integral using a finite number of randomly generated rays. While

these rays can be generated following a uniform distribution, it is common to instead perform importance sampling, preferring rays that align more with the surface normal $\boldsymbol{n}$. The reason stems from the $(\omega_i \cdot \boldsymbol{n})$ term in the rendering equation: rays that are aligned more with the normal contribute 'more' to the final integral than those perpendicular. Using Monte Carlo methods with importance sampling essentially means that fewer rays are needed to obtain an accurate estimate compared to using a basic uniform distribution. We will see the influence of Monte Carlo methods in the implementation of voxel cone tracing.

2.5.2 Raymarching

Consider wanting to render the effects of participating media: some type of semi-transparent object, such as dust or a cloud. With raytracing, intersections occur at discrete points between the casted ray and the object it hits. Therefore straightforward raytracing will not work, as there is no single intersection when dealing with participating media. Raymarching is an extension of raytracing that involves sampling multiple points along the casted ray in defined intervals. The values sampled along the ray are accumulated and blended based on opacity.

Chapter 3

RELATED WORK

Global illumination is a broad topic that can be approached in many ways. With respect to this thesis, we only focus on comparisons with other real-time, fully dynamic techniques. Many of these techniques borrow similar ideas and build off of each other. The implementations for a single technique can also vary greatly based on data structures used, shading models, and other considerations. The different tradeoffs made and how they affect the final implementation are important to understand, as there is often not a single best way to approach the problem of global illumination. Here we present some of the more popular and relevant methods of achieving full global illumination and attempt to note the major differences between them and our implementation.

3.1 Reflective Shadow Maps

Dachsbacher and Stamminger introduced the idea of Reflective Shadow Maps (RSMs) in 2005 [14]. Their work extends traditional shadow mapping to support single bounce indirect illumination. The main idea is to treat each pixel in the shadow map as a virtual point light (VPL)—i.e., each pixel in the shadow map corresponds to a directly lit point—which illuminates the rest of the scene[1]. The information gathered for each reflective shadow map is shown in Figure 3.1. In order to efficiently compute the contribution of the VPLs for a point in the scene a fixed number of samples are taken from the shadow map. They also apply an interpolation scheme to reduce computation on smooth parts of the scene.

[1] VPLs were introduced in Keller's work on Instant Radiosity [26].

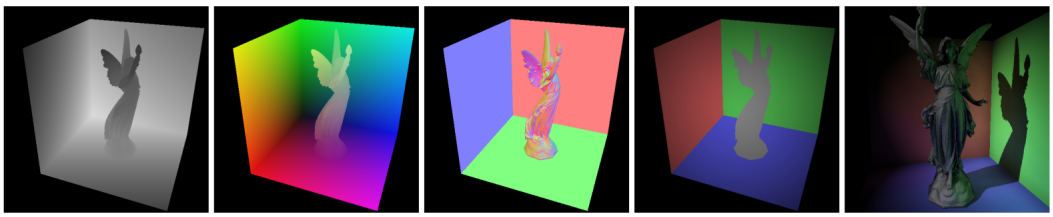


Figure 3.1: The different components of a reflective shadow map (depth, world coordinates, normal, and color) and the resulting image [14].

A notable downside of this method is occlusion information is not accounted for: a point y could contribute indirect lighting to another point x even if there is other geometry blocking the path between x and y. The authors apply a separate ambient occlusion pass to partially mitigate this problem. The method is also only designed for low frequency lighting details. Also, while directional, point, and spot lights should theoretically work, objects that themselves emit light were not addressed.

3.2 Cascaded Light Propagation Volumes

Light Propagation Volumes (LPVs) are another method for approximating low frequency indirect lighting, developed by Kaplanyan and Dachsbacher in 2010 [24]. The method relies on an iterative based light propagation algorithm within a volumetric grid structure. This structure, the LPV, is initialized by injecting VPLs into its cells—for example, by using RSMs. Next, the light is propagated multiple times between adjacent cells; each propagation results in the LPV having a more complete representation of the indirect light in the scene.

LPVs improve on the basic RSM idea by filtering the light in multiple steps. It also stores low resolution geometry information for occlusion purposes. They also tackle the issue of handling large scenes with the concept of cascaded LPVs: multiple individual LPV structures are used with different sizes, focusing detail near the camera (shown in Figure 3.2).

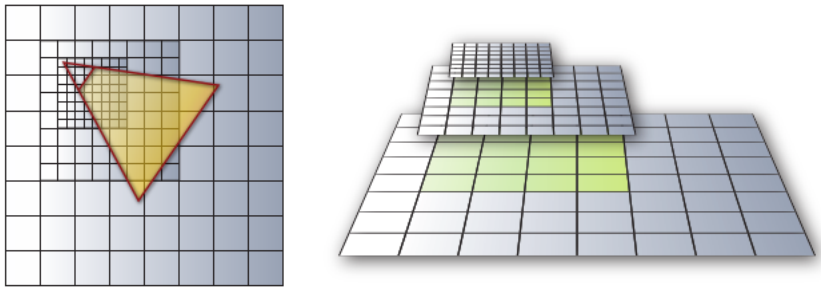


Figure 3.2: A diagram demonstrating the grids used for cascaded LPVs [24].

The main drawback of this method is the lighting is low frequency only. Proper glossy (specular) reflections are still wanted in many applications. Also, the geometry information obtained is not complete: it is created by reusing other results from shadow mapping and depth peeling.

3.3 Rasterized Voxel-Based Dynamic Global Illumination

Rasterized Voxel-Based Dynamic Global Illumination described by Doghramachi [16] is based heavily on LPVs but uses a voxelized scene representation for the initial steps of LPV creation. This avoids some of the difficulties when injecting VPLs compared to LPVs. In addition, the geometry information made available by voxelization is complete (although low-resolution) and does not require external inputs like shadow maps or depth peeling.

Just like LPVs, the main downside here is the lack of specular reflections, which stems from the low resolution grid used for the LPV (32x32x32, as mentioned in the implementation, whereas we aim for somewhere on the order of 128^3 to 256^3). Also, the method does suffer from some light leaking due to how point lights are injected into the scene: instead of creating an RSM (which handles occlusion properly), all voxels within a defined distance of the point light are injected into the initial LPV.

3.4 Voxel Cone Tracing

In 2011, Crassin et al. introduced Interactive Indirect Illumination Using Voxel Cone Tracing [13]. A primary improvement over the existing methods sought in this method was to obtain higher frequency lighting details, such as specular reflections. The core idea to balance performance with lighting detail is to make use of a filtered radiance representation which is then sampled using voxel cone tracing, a concept they introduce.

The first step in the algorithm is to create a high resolution radiance representation. Crassin et al. inject radiance into the leaves of an octree using the familiar RSM technique. An octree is used to reduce memory consumption, a consequence of using such a high resolution radiance representation. The octree is then filtered recursively bottom up until all inner nodes in the octree have a filtered radiance value. The final step, voxel cone tracing, is essentially a combination of raymarching and mipmapping: raymarched samples are taken from successively lower levels of detail. An illustration of this is shown in Figure 3.3.

Although the octree representation does help alleviate issues when dealing with sparse scenes, GPU memory consumption is still an issue especially for larger scenes. In addition, due to their tree-like nature, octrees are awkward for GPU use and do not benefit from things like hardware supported texture filtering. To address this, NVIDIA decided to use a clipmap instead of an octree in their standard implementation of voxel cone tracing, VXGI [2].

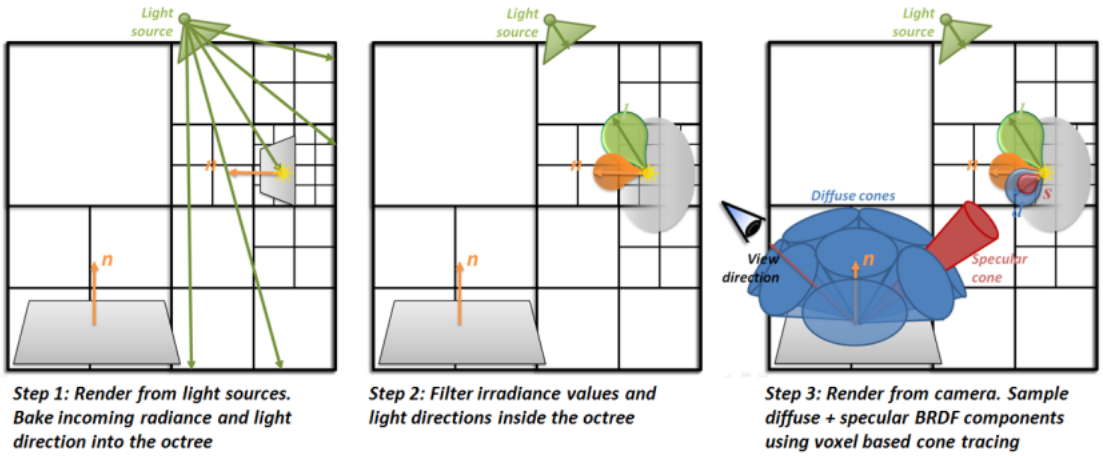


Figure 3.3: An illustration of the voxel cone tracing algorithm by Crassin et al. [13].

Chapter 4

IMPLEMENTATION

At its core, the application we created for this thesis is a standard forward rendering engine supporting multiple lights, shadow mapping, and normal mapping—in addition, of course, to full real-time dynamic global illumination. The scene to be rendered is composed of one or more actors, which are simply meshes loaded from the generic Wavefront OBJ file format. Actors are also capable of rigid body animations.

The rendering pipeline can be broken down into several render passes, each of which will be explained in its corresponding section. In general, the main steps taken are generating a voxelized representation of the scene, creating a filtered representation of light from the voxels and light sources, and, finally, shading the scene.

The application itself is written in C++11 and uses OpenGL as the graphics API. In order to make use of modern graphics features like compute shaders and direct state access we require OpenGL version 4.5 (released in 2014). The project uses CMake as its build system and has been tested on Linux and Windows.

4.1 Voxelization

The goal of voxelization is to create a sparse 3D representation of the geometry in the scene. The information associated with each voxel is stored in two 3D textures: one for color and opacity and another for surface normal. We refer to these textures as `voxelColor` and `voxelNormal`, respectively. Each texture has an internal format of `RGBA16F` or `RGBA8`, depending on hardware capabilities. The resolution of the textures is configurable at runtime, although the default is 256^3.

The extents of our voxelization volume—the area in world space that will end

up inside the voxel textures—is also configurable and by default is set to enclose the entire scene. This volume can be static or can track the camera (with the camera being at the center of the textures). In addition, to prevent some issues with temporal artifacts, the voxel textures are snapped to a discrete grid corresponding to the size of each voxel cell in world space.

We present two different approaches for scene voxelization: one utilizing hardware rasterization and the other hardware tessellation.

4.1.1 Rasterization-Based Approach to Voxelization

The first method for voxelization utilizes the GPU rasterization pipeline and is based on [11]. The voxelization is performed completely in a single render pass. The basic idea is to rasterize the scene such that each fragment generated by the GPU corresponds to a single voxel in the 3D texture.

One of the main challenges with voxelization is ensuring there are no holes or cracks in the resulting grid. Two techniques are used to mitigate this (and will be covered in-depth in the following paragraphs). First, for each triangle we project along the triangle's normal's dominant axis. This maximizes the number of fragments generated for a given triangle. Second, we perform conservative rasterization. Normally, the GPU only generates a fragment for a given pixel if a triangle overlaps the center of the corresponding pixel. This results in cracks in the voxelized grid since even if a triangle is inside a voxel it may not be rasterized.

Before the draw call, the necessary matrices to project the geometry into the voxelization volume are created. The projection matrix is orthographic, with bounds corresponding to the extents of the voxelization volume. In order to project along each triangle's dominant axis, we also need three view matrices: for each axis a view matrix is constructed where the view position is at the edge of the voxel volume and

looks down its respective axis. The final matrix used to transform a point in world space to a point in the voxel texture for a particular axis is composed by multiplying the orthographic matrix with the appropriate view matrix.

For the shader program, the voxel textures are bound as `image3D` objects (as textures are read-only in shaders). Following from that, since the framebuffer is not being written to, color writing is disabled. Lastly, depth testing and depth writes are also disabled, since the goal is to voxelize the entire scene. The algorithm is shown in Algorithm 1 and explained in detail below.

Algorithm 1 Voxelization

```
for each triangle t do
    find dominant axis of t using surface normal
    apply projection matrix corresponding to the dominant axis
end for
Rasterization                                          ▷ Performed by hardware
for each rasterized fragment do                        ▷ position is in NDC
    transform NDC coordinates to texture space ([0, 1])
    Realign NDC coordinates with standard coordinate system
    Insert fragment into voxel texture
end for
```

Starting in the vertex shader, the only operation performed is transforming the object from object space to world space using its appropriate model matrix. In the geometry shader we take a single triangle in world space, transform it based on its dominant axis, and output the resulting triangle in clip space. The triangle's dominant axis is determined by finding the largest component of its normal (by absolute value)[1]. We then apply the appropriate projection matrix according to the dominant axis

[1]Finding the dominant axis is really finding the maximum dot product of each axis with the normal, but since we use the standard basis this reduces to just finding the largest component of the normal.

and output the resulting triangle to be rasterized. In the fragment shader, each fragment's position in Normalized Device Coordinates can be used to determine its position within the voxel textures. Note, however, that the components of each fragment's position need to be realigned due to projecting along different axes. Finally, the voxel information can be written into the voxel textures, which are bound as `image3D` objects in the shader. The result of this voxelization is shown in Figure 4.1.

However there is still one more issue: multiple voxel fragments can be mapped to the same cell in the voxel textures. Without any synchronization, the final voxel values are not deterministic and can result in temporal artifacts. To solve this, we average the resulting color value using atomic operations[2].

Figure 4.1: Visualization of the voxels resulting from the rasterization-based approach (without conservative rasterization).

[2]Other approaches to this problem (with various tradeoffs) can be used (Doghramachi uses an atomic max operation on a custom defined metric in order to avoid needing to perform an average [16]). These are generally used to combat performance issues or hardware limitations.

Conservative Rasterization

As mentioned previously, we need conservative rasterization to minimize holes and cracks in the voxelization. To achieve this, we use an MSAA-based approach [45]. MSAA (multi-sample anti-aliasing) is a technique used for smoothing out ragged edges (aliasing) in a rasterized image. Instead of only sampling at one position inside a pixel to determine if a fragment should be generated multiple samples are used (hence the name). These samples are typically distributed to maximize the possibility of a rasterized fragment being produced. If any of the samples overlap with the triangle a fragment is generated for that pixel. An illustration of MSAA is shown in Figure 4.2. Applied to voxelization, this means there is a smaller chance that a triangle will fail to produce a fragment for any voxel it only partially occupies. While the MSAA method of conservative rasterization is not perfect (a partially occupied voxel won't always produce a fragment), it is cross-platform, simple to use, and efficient. Other notable methods would be manually dilating each triangle in a geometry shader (better quality but slower) [4] or using a GPU vendor specific extension, like `GL_NV_conservative_raster` (better quality but not cross-platform). A comparison between no conservative rasterization, MSAA-based conservative rasterization, and the NVIDIA extension is shown in Figure 4.3[3].

4.1.2 Tessellation-Based Approach to Voxelization

The other method for voxelization relies completely on hardware tessellation: no rasterization required. Instead of generating fragments for each voxel we instead attempt to generate vertices for each voxel. One benefit of this method is the mapping from vertex to voxel position is fairly straightforward and does not require

[3] Also note the erroneous fragments generated when using the NVIDIA extension. These extra fragments are produced by some methods of (overly) conservative rasterization and the user must manually cull these fragments.

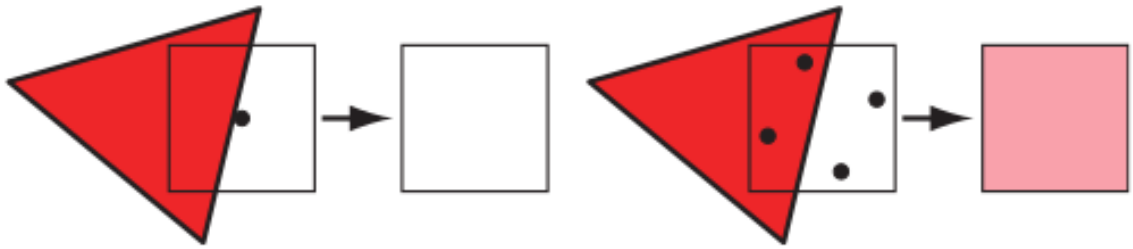


Figure 4.2: With MSAA, multiple points within a pixel are used to determine whether a fragment should be generated [5].

multiple projection matrices like in the rasterization-based approach. Fei et al. [18] also developed an algorithm similar to ours using tessellation.

The core part of this approach is determining the appropriate tessellation level for a given triangle. Recall that this step is the responsibility of the tessellation control shader. Afterwards, tessellation primitive generation produces the tessellated vertices and provides them to the tessellation evaluation shader, where we compute each vertex's position in the voxel texture and store the vertex's color. The final voxelized scene is shown in Figure 4.4.

The desired level of tessellation will produce one vertex for each voxel the triangle covers. Therefore, for a particular triangle, the tessellation level is determined based on the triangle's size in world space. Recall that for triangular tessellation patches there are four values: three outer tessellation levels (one for each edge) and one inner tessellation level. To determine the outer tessellation level for an edge E, we effectively determine how many voxels E passes through (accounting for the direction of E): $outerLevel = \frac{|E|}{(\frac{E}{||E||} \cdot voxelSize)}$. To determine the inner tessellation level, we find the maximum altitude of the triangle and then perform a similar computation as with the outer level. With a, b, and c being the lengths of the triangle edges, the altitude from a side $x \in \{a, b, c\}$ is given by

$$h_x = \frac{2\sqrt{s(s-a)(s-b)(s-c)}}{x}$$

(a) No conservative rasterization

(b) MSAA-based conservative rasterization

(c) NVIDIA's conservative rasterization extension

Figure 4.3: Conservative rasterization produces 'extra' fragments in order to produce a more solid voxelization. The effects are particularly noticeable on the pillar between the red and green curtain on the left.

where $s = (a + b + c)/2$ is the semiperimeter and the square root term is the triangle's area using Heron's Formula. The maximum altitude is then $h_{max} = \max(h_a, h_b, h_c)$. Finally, we have $innerLevel = h_{max}/voxelSize$.

After specifying the tessellation levels in the tessellation control shader, tessellation primitive generation produces the vertices that correspond to the voxels. In the tessellation evaluation shader the vertex position (in world space) is used to determine the final voxel position and is written out to the voxel texture.

An important consideration for this method is the maximum tessellation level supported by the hardware. For large triangles it is possible the tessellation will not produce a solid voxelization. A simple way to address this issue is to break large triangles into smaller ones when loading the mesh. Another consideration is for small triangles where all vertices occupy a single voxel. To reduce the number of writes to the voxel texture the contributions of these vertices can be consolidated in the shader into a single operation.

Figure 4.4: Scene voxelized using tessellation-based voxelization.

4.2 Shadow Mapping

Shadow mapping is a well known technique used for efficiently creating shadows [48]. Similar to the related works, the resulting shadow map is also treated similarly to an RSM for radiance injection.

To generate a shadow map for a light, we render the scene from the light's perspective and gather the depth values of the fragments. Since we need the GPU to write to an arbitrary texture (the shadowmap), we create a framebuffer object (FBO) and attach the texture as a depth attachment[4]. The FBO does not require any color attachments, as we are only interested in the depth. The matrix used to transform vertices from world space to light space is an orthographic projection matrix multiplied with a view matrix generated from the light. This matrix is often called the light view matrix. Since the GPU will automatically write depth values, the fragment shader can be completely empty. The resulting shadow map contains floating point depth values (in the range [0, 1]) which represent the distance of each point visible to the light itself. Also note that these depth values are linear since an orthographic projection matrix is used: if a perspective projection matrix is used, the depth values scale logarithmically. Figure 4.5 shows the contents of a shadowmap for our scene.

The only difference between generating a shadow map and a complete RSM is to also store the diffuse color and normal along with the depth values. Since our implementation stores voxel colors and normals already, a full RSM isn't necessary for injecting the VPLs into the radiance texture.

[4]This general technique is referred to as render-to-texture.

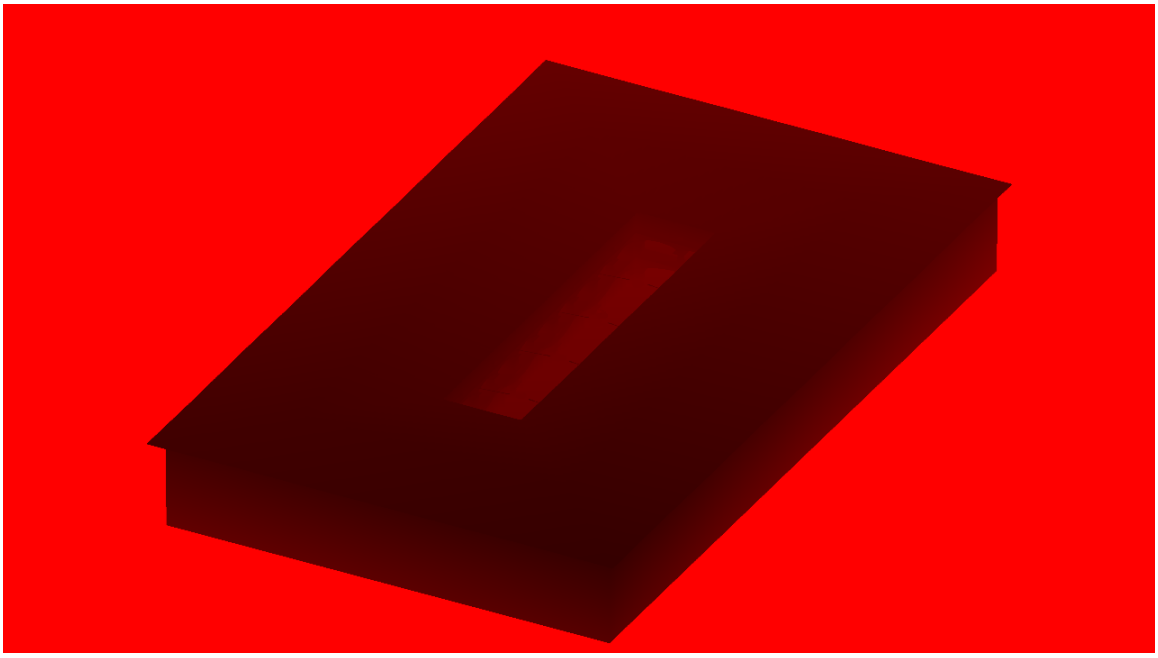


Figure 4.5: An example of a shadowmap with the depth value mapped to the red component of the image.

4.3 Radiance Injection

In this render pass, we fill a 3D texture with virtual point lights from the shadow map. Recall that the VPLs are light sources which represent the light being bounced off of geometry within the scene. Following from RSMs, the points at which this happens are precisely where the light hits the geometry, which is stored in the shadow map. Therefore, this pass involves taking all points in the shadow map and projecting the color of the corresponding geometry into the radiance texture.

The radiance texture, `voxelRadiance`, has dimensions equal to that of the other voxel textures. It has a format of `RGBA8` and stores a color value and opacity.

The most straightforward way to accomplish radiance injection from the shadow map is to use a compute shader and launch one thread for each pixel in the shadow map. Each thread first reads the depth value at its respective pixel, which can then be used to reconstruct the pixel's position in light space. Then, by using the inverse

light view matrix, we project it from light space to world space. The world space position is used to calculate the voxel position, which is used to sample from the voxel textures and finally write the color into the radiance texture, as seen in Figure 4.6.

It is important that the shadowmap is rendered with a high enough resolution (relative to the voxel texture resolution) to ensure a smooth radiance injection. If the shadowmap resolution is too low, there will be gaps between voxels which cause lighting artifacts. Note also that we do not need to be concerned with synchronization and atomic operations in the case that multiple threads access the same voxel position since we are only interested in the base color.

In addition to injecting the VPLs, we also transfer occlusion information (opacity) stored in the completely voxelized scene into the radiance texture. This operation is a straightforward transfer accomplished using another compute shader.

Algorithm 2 Radiance Injection

for each pixel coordinate $p = (p_x, p_y)$ in shadowmap **do** ▷ $p_x, p_y \in [0, 1]$

  $depth = \text{sampleDepthFromShadowmap}(p)$

  $ndc = (p_x, p_y, depth) * 2 - 1$ ▷ Convert coordinate to NDC

  $worldPosition = inverseLight * ndc$

  $voxelPosition = \text{computeVoxelPosition}(worldPosition)$

end for

4.4 Radiance Filtering

With the highest level of detail of the radiance texture filled, the next step is creating the filtered representation. Each successive level is half the resolution of the previous level. Therefore, the maximum number of levels the texture may have is $log_2(\max(\text{width}, \text{height})) + 1$.

OpenGL is able to automatically generate mipmaps for 3D textures by calling

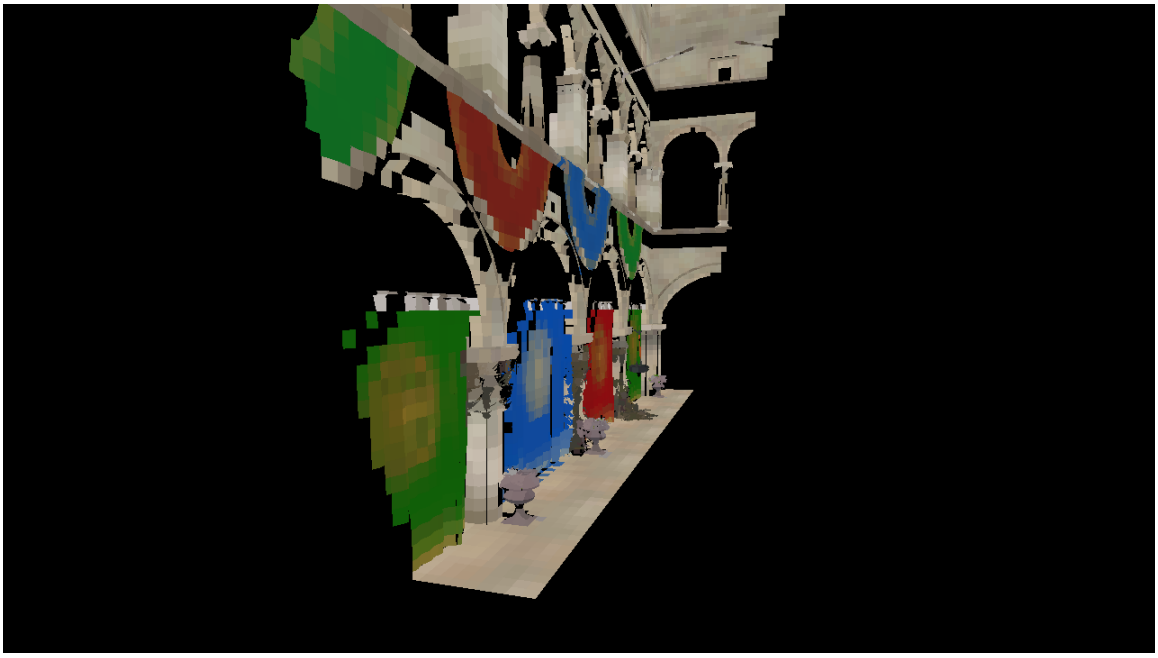
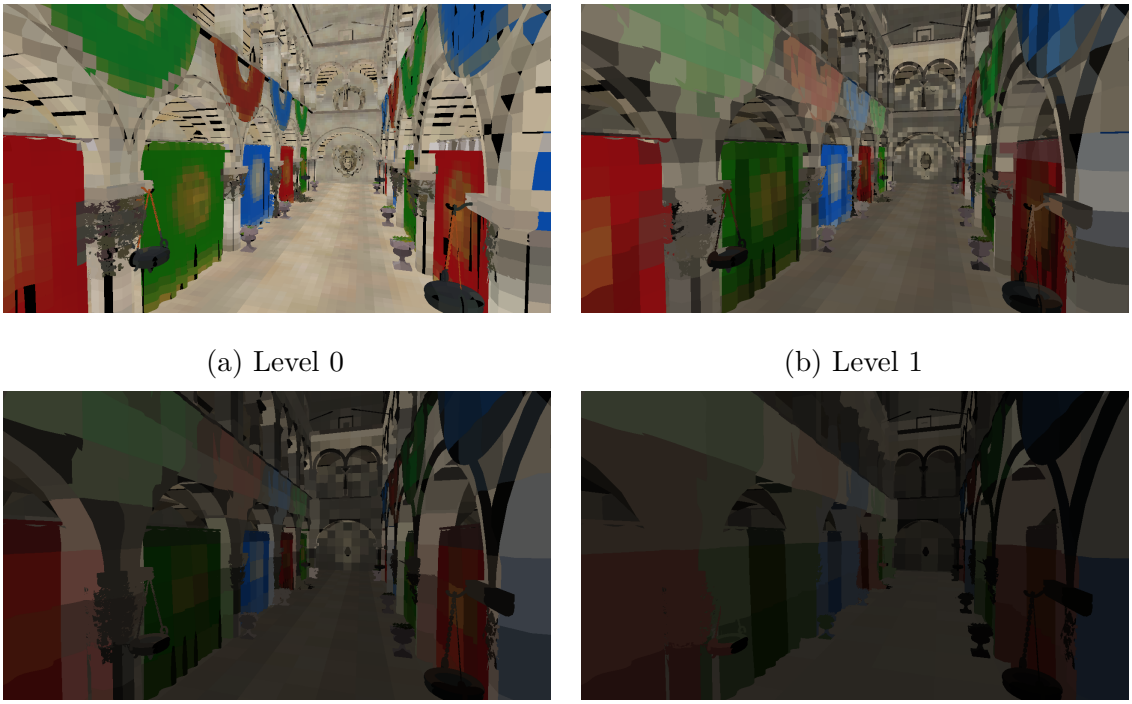


Figure 4.6: The radiance texture injects VPLs according to the shadowmap: only voxels hit by the light source result in a VPL.

`glGenMipmaps(GL_TEXTURE_3D)`, however manually filtering the textures using a compute shader turned out to be much faster for this application (how OpenGL calculates mipmaps is implementation defined, so the performance can vary between systems and drivers). To perform the filtering, a compute shader kernel is launched for each level of the radiance texture (not including the base level). The kernel is launched with one thread for each voxel in the current level. Each thread can be conceptually located at the corners of each voxel in the previous level and average the surrounding 8 values to compute the filtered value of its respective voxel.

The other benefit to manually filtering is that different methods can be used. For example, adjusting the size or weights of the filtering kernel could result in different effects.



(a) Level 0 (b) Level 1

(c) Level 2 (d) Level 3

Figure 4.7: The first 4 levels of the radiance texture are shown here (using `GL_NEAREST_MIPMAP_NEAREST` filtering for demonstration purposes).

4.5 Shading

The final shading step takes all of the previously generated information and renders the scene. Direct lighting is computed from multiple lights and uses the physically based Cook-Torrance shading model [9]. The classic Blinn-Phong shading model [40] is also supported. Indirect lighting is computed using voxel cone tracing. Normal mapping (similar to bump mapping [6]), shadow mapping, and post processing effects also occur here. A complete outline of computing the final shaded color is shown in Algorithm 3.

Two important inputs for the fragment shader are material information and light information. The material information for each fragment is provided in a struct `Material`, uploaded as a uniform buffer object, and contains color and texture

information. Lights are stored as an array of struct `Light`s and contain information associated with them such as light type (e.g. directional or point light), position, and color. The array is uploaded as a shader storage buffer object, which allows the array to have a dynamic size queryable at shader runtime.

4.5.1 Direct Lighting

We calculate direct lighting by iterating over each light source in the scene and summing all lighting contributions. We compute diffuse and specular lighting according to the Cook-Torrance shading model [9] (without the ambient term, which is computed by the voxel cone tracing):

$$r = \sum_{i=0}^{n} l_{i_c} * (\boldsymbol{n} \cdot \boldsymbol{l_i}) * (d * r_d + s * r_s)$$

where r is the resulting color, n is the number of lights, l_c is the light color, $\boldsymbol{n}$ is the surface normal, $\boldsymbol{l}$ is the light vector, d and s are scaling factors, and r_d and r_s are the diffuse and specular components (the BRDFs). The diffuse component r_d is calculated with the Lambertian model for diffuse light, which is simply c/π, where c is the material's surface color. The specular component is

$$r_s = \frac{DGF}{\pi(\boldsymbol{n} \cdot \boldsymbol{l})(\boldsymbol{n} \cdot \boldsymbol{v})}$$

where D is a Normal Distribution Function, G is the Geometric Attenuation Function, F is the Fresnel term, $\boldsymbol{n}$ is the surface normal, $\boldsymbol{l}$ is the light vector, and $\boldsymbol{v}$ is the view vector. The Cook-Torrance model allows for different choices of D, G, and F. Our implementation uses the Trowbridge-Reitz GGX Normal Distribution Function, Smith Schlick-GGX Geometric Attenuation Function, and Schlick's approximation for the

Algorithm 3 Final Shading (performed on each rasterized fragment)

$directLighting = 0$

for each light in the scene **do**

$directLighting = directLighting + \text{computeDirect}()$ ▷ Diffuse and specular lighting

if in shadow **then**

$shadowFactor = \text{computeShadowAmount}()$

$directLighting = shadowFactor * directLighting$

end if

end for

$indirectLighting = 0$

for each diffuse cone **do**

Transform cone direction to world space ▷ Using TBN matrix

$color, occlusion = \text{coneTrace}()$ ▷ Perform cone tracing, resulting in a color and occlusion factor

$indirectLighting = indirectLighting + occlusion * color$

end for

$reflectedDirection = \text{reflect}()$ ▷ Compute direction of reflected ray

$reflectColor, reflectOcclusion = \text{coneTrace}()$

$indirectLighting = indirectLighting + reflectOcclusion * reflectColor$

$finalColor = directLighting + indirectLighting$

$finalColor = finalColor/(finalColor + 1)$ ▷ Tone map

$finalColor = finalColor^{\frac{1}{2.2}}$ ▷ Gamma correction

Fresnel term[5]:

$$
\begin{aligned}
D(\boldsymbol{n}, \boldsymbol{h}, \alpha) &= \frac{\alpha^2}{\pi((\boldsymbol{n} \cdot \boldsymbol{h})^2(\alpha^2 - 1) + 1)^2} \\
G1(\boldsymbol{n}, \boldsymbol{v}, k) &= \frac{\boldsymbol{n} \cdot \boldsymbol{v}}{(\boldsymbol{n} \cdot \boldsymbol{v})(1 - k) + k} \\
G(\boldsymbol{n}, \boldsymbol{v}, \boldsymbol{l}, \boldsymbol{k}) &= G1(\boldsymbol{n}, \boldsymbol{v}, k) G1(\boldsymbol{n}, \boldsymbol{l}, k) \\
F &= F0 + (1 - F0)(1 - \cos\theta)^5
\end{aligned}
$$

where α is material roughness, $\boldsymbol{h}$ is the half vector, $k = \frac{(\alpha+1)^2}{8}$, θ is the angle between $\boldsymbol{h}$ and $\boldsymbol{v}$, and F0, the surface reflection at zero incidence, is 0.04 for dielectrics and the surface albedo for metals[6].

4.5.2 Indirect Lighting (Voxel Cone Tracing)

To compute the indirect lighting at a given point, we perform voxel cone tracing. Recall that to compute indirect light we are effectively approximating a surface integral over a hemisphere. With voxel cone tracing, this integral is reduced to summing the contributions of several cones, each of which are defined by a direction and cone angle[7]. To compute the integral perfectly would require infinitely many cones. Instead, we use six cones: five for diffuse light and one for a specular highlight[8].

For the diffuse cones we chose an aperture of 45° with one cone oriented in the direction of the surface normal and the others evenly distributed around the normal and tilted up at an angle of 45°. We also assign weights to each cone based on a uniform distribution. Note that in order to compute the orientation of each cone in

[5]These choices for D, G, and F are the same as in Unreal Engine 4 [25].

[6]This is not completely physically accurate but is a good and fast approximation.

[7]Note that as the cone angle approaches zero we effectively have a ray. In fact, the main difference between classic raymarching and cone tracing is the cone angle (which is used to determine which level of the filtered radiance to sample from).

[8]The number of cones and their associated directions and angles are determined experimentally.

world space we must multiply the chosen cone directions by the TBN matrix, since the cone directions are defined in tangent space. To get the final diffuse contribution, the cone tracing is performed for each cone and the values are weighted and summed appropriately.

The specular cone is oriented in the direction of the reflection vector, which is calculated as $-\boldsymbol{v} - 2 * (\boldsymbol{n} \cdot -\boldsymbol{v}) * \boldsymbol{n}$, or by using the GLSL function `reflect()`. The aperture is derived from the material's roughness according to the following formula[9]: $0.1 * \pi * roughness$.

The radiance texture is sampled at varying levels of detail in order to approximate the amount of indirect light at that point. At it's core, voxel cone tracing is raymarching through a mipmapped texture. In order to determine which miplevel to sample from, the idea of cone tracing is introduced. Instead of having a simple ray, we imagine a cone with a particular aperture (angle). The cone's height is analogous to a ray's length and the size (diameter) of the cone's base grows as the height increases. We can then map the diameter to a level of detail.

Combining this all together, we sample from the radiance texture at a level of detail related to the height of the cone at our sampling point. In this way, samples close to the start of the cone come from higher detailed data and samples far from the start come from lower detailed data. This allows the voxel cone tracing to gather both high frequency and low frequency details of the indirect light.

The implementation of a single cone tracing instance is done entirely within a fragment shader. The main inputs are the radiance texture, the position to start tracing from, and the direction in which to trace. There are also several configurable parameters that influence the cone tracing. We have `steps`, the maximum number of samples to take; `bias`, the initial offset from the starting position (in order to avoid

[9]Determined experimentally.

self illumination); `coneAngle`, the angle of the cone; `coneHeight`, the starting height of the cone; and `lodOffset`, a constant offset used when determining the level of the radiance texture to sample from. Also, since the cone tracing is performed in texture space, a `scale` is applied to normalized direction vectors. For each step, the cone's radius is computed from the cone's height and angle. From this, we compute the level of detail to sample from. After getting the sample color and alpha value a forward blending scheme is used. Finally, the cone's height is increased for the next iteration of the loop. The complete shader code for cone tracing is shown in Listing 4.1. The lighting contribution from the indirect light (both diffuse and specular) is displayed in Figure 4.8 and the occlusion values are displayed in Figure 4.9.

```
vec4 traceCone(sampler3D voxelTexture, vec3 position, vec3 normal, vec3
    direction, int steps, float bias, float coneAngle, float coneHeight,
     float lodOffset)
{
    vec3 color = vec3(0);
    float alpha = 0;

    float scale = 1.0 / voxelDim;
    vec3 start = position + bias * normal * scale;
    for (int i = 0; i < steps && alpha < 0.95; i++) {
        float coneRadius = coneHeight * tan(coneAngle / 2.0);
        float lod = log2(max(1.0, 2 * coneRadius));
        vec3 samplePosition = start + coneHeight * direction * scale;

        vec4 sampleColor = textureLod(voxelTexture, samplePosition, lod
            + lodOffset);
        float a = 1 - alpha;
        color += sampleColor.rgb * a;
        alpha += a * sampleColor.a;
        coneHeight += coneRadius;
```

```
    }

    return vec4(color, alpha);
}
```

Listing 4.1: Shader function for performing voxel cone tracing.

4.6 Voxel Warping

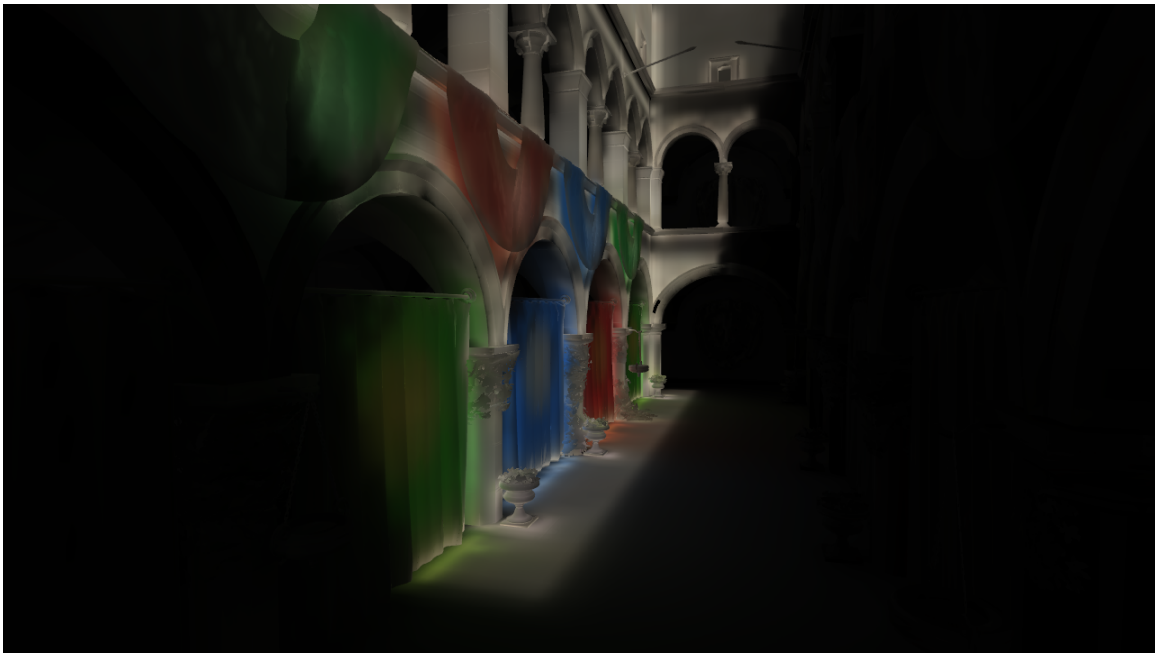
The goal of voxel warping is to vary the density of the voxel resolution according to some metric. This effectively increases the detail of the voxelization, leading to better global illumination. We present two different approaches for voxel warping: warp functions and perspective warping.

4.6.1 Using a Warp Function

Previously, voxel positions were determined linearly from their position inside the voxel volume to a position in the voxel texture. This approach takes that linear position and applies a mapping to it to find the final position inside the voxel texture. With an appropriate warping function, the voxel density can be modified. For this approach, the warping function's job is to increase detail near the camera and then smoothly fall off for objects further away.

Linear Warp Function

Recall from voxelization that in order to determine the location in the voxel texture for a given fragment we perform a linear mapping from world space to image space. As an intermediate step, we have the voxel position in texture space (in the range $[0, 1]$). Thus, for example, if the position along the y axis is 0, it will go into the



(a) Diffuse indirect light

(b) Specular indirect light

Figure 4.8: The diffuse and specular contributions from cone tracing (before multiplying by occlusion).

bottom of the voxel texture and if 1, it will go into the top part of the voxel texture.

Now, consider mapping the coordinate in texture space to so-called warp space, which

Figure 4.9: Occlusion values resulting from voxel cone tracing. The occlusion is multiplied with the cone traced lighting in order to account for occlusion of nearby objects (i.e. voxel based ambient occlusion).

is also in the range $[0, 1]$. Without any modifications, this is a straight line with a slope of 1, representing a linear 'warping' function. If we let $w_{linear} : [0, 1] \rightarrow [0, 1]$ be the warping function and $x \in [0, 1]$ be the coordinate in texture space then we have $w_{linear}(x) = x$. Of course, this warping function does not do anything useful. For that, we need a different, nonlinear, warping function.

Nonlinear Warping

Consider the function $w_{logistic}(x) = \frac{1}{1+e^{-x}}$ (a basic logistic function, a type of "S"-shaped curve). Notice how as x approaches 0 or 1 the slope decreases and in the middle the slope is greater than one. If we draw lines up and over from the curve for $x = 0.4$ and $x = 0.6$ we see that the range of $w_{logistic}(x)$ is greater than that of $w_{linear}(x)$. In other words, the positions within those particular x values 'take up more space' in the voxel texture. Likewise, towards the ends, the slope decreases towards

zero and thus takes up less space. This achieves the goal of varying the voxel density resolution according to a simple function. We also see that the slope, $w'(x)$, represents the voxel density: if $w'(x) = 1$, the density is the same as the linear mapping; if $w'(x) > 1$, the density is greater than the linear mapping; if $w'(x) < 1$, the density is less than the linear mapping.

The ideal warping function increases voxel density near the camera and decreases voxel density further away. The logistic function provided does this[10], however there are some issues. Recall from Voxelization (section 4.1) that, with the rasterized approach, the scene is rendered with a viewport of dimensions equal to that of the voxel texture. The discretized fragment positions will therefore all be in step sizes corresponding to this viewport resolution. When the warp function is applied where $w'(x) > 1$, we can run into issues where adjacent fragments will 'skip' a position in the voxel texture. In essence, the voxel fragments are not generated with fine enough resolution to smoothly transition after being warped [11]. To resolve this, the scene must be voxelized with a viewport resolution scaled by the maximum derivative of $w(x)$. In design terms, this means choosing a warping function with a steep slope will result in needing to voxelize the scene with a larger viewport resolution, which can hurt performance. Figure 4.10 demonstrates this issue.

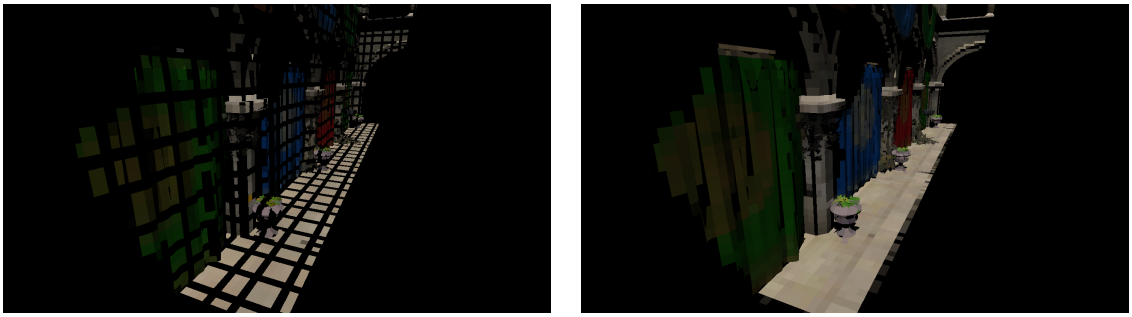


Figure 4.10: Cracks in the voxels caused by higher voxel density (left) and filling the holes by voxelizing with a higher resolution (right).

[10]The camera is at the center of the voxel grid, so it corresponds to $x = 0.5$ in the warping function.

[11]This is similar to the concept of the Nyquist Frequency, where we are not sampling the signal at a high enough rate

Using the logistic function as the warping function also has another issue: the slope at the ends approaches zero. This leads to a large portion of the voxelized region ending up in relatively few voxels, which diminishes the accuracy of the voxelization greatly. Instead, we want to place a lower bound on $w'(x)$. The solution to this is to use a cubic spline, i.e. $w(x) = a + bx + cx^2 + dx^3$. Then, with α as the desired end slope, we use the constraints $w(0) = 0, w(1) = 1,$ and $w'(0) = w'(1) = \alpha$ and solve for the variables a, b, c, and d. This gives the warping function: $w(x) = \alpha x + (3 - 3\alpha)x^2 + (2\alpha - 2)x^3$. For our implementation, $\alpha = 0.25$.

4.6.2 Perspective Warping

The other method of warping implemented is based on perspective projection. Recall that when transforming coordinates from view space to clip space we use a projection matrix. To emulate the visual effect of distant objects appearing smaller, a perspective projection matrix applies a scaling factor to any transformed position coordinate. This scaling effectively causes distant objects to occupy less area in screen space in the final rendered image. Similarly, for our perspective warping, we use a perspective matrix to manipulate the final voxel position.

To determine the voxel position from a position in world space, we apply both a perspective projection and view matrix. This results in a point in NDC corresponding to its position within the view frustum[12]. This value is then shifted and scaled to be in texture space, which is the voxel position.

A visualization of the voxels resulting from perspective warping is shown in Figure 4.11. The view frustum is divided into voxels where voxels closer to the camera end up being smaller and those further away are bigger.

[12]We also must linearize the depth (z) component of the position, as a perspective projection causes the depth to be logarithmic.

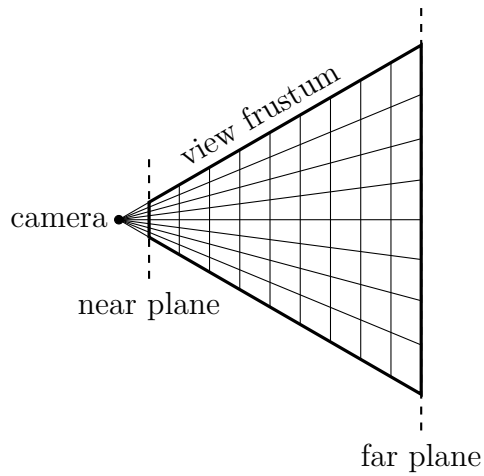


Figure 4.11: Diagram of the voxels resulting from perspective warping.

4.7 Miscellaneous

4.7.1 Depth Prepass

With voxel cone tracing, each fragment shader invocation is quite expensive. If we have overlapping objects in the scene then fragments will be shaded for each pixel, but only one will be the final color. To ensure we only shade the final fragment, we have a prepass which rasterizes the scene and only writes the depth value. This operation is extremely quick. Now, the depth buffer contains only the depth of the fragment that will end up on the screen. When the full shading pass is performed all fragments that fail the depth test can be immediately discarded, avoiding the expensive voxel cone tracing[13].

4.7.2 Temporal Filtering

Typically the radiance texture is cleared completely at the beginning of each frame. Due to the discrete nature of the voxel grid, moving objects and lights can

[13]This technique is only needed for a forward rendering engine: for a deferred rendering engine the final depth values are calculated when generating the G-buffer (geometry buffer).

cause temporal artifacts (flickering) as the voxels are updated. These artifacts can be improved by incorporating previous radiance values into the current frame's radiance. To do this, we can blend some factor of a voxel's previous radiance value with its new value[14].

[14]This factor is configurable at runtime.

Chapter 5

RESULTS AND DISCUSSION

Here we examine both the performance and visual quality of our global illumination algorithm. The primary means of evaluating performance is based on frame time. The times of each render pass are recorded as well as a total frame time. Evaluating visual quality is based largely on whether the lighting appears smooth and believable with a minimal amount of noise and artifacts.

We also discuss the application of voxel warping and how it compares with other global illumination methods. Unfortunately, direct performance and visual quality comparisons are difficult to objectively measure as there are many contributing factors—mesh complexity and optimizations, texture resolution and format, shading model, graphics API, hardware, GPU driver version, and more—that affect the final comparison, in addition to the exact implementation details. Nevertheless, we provide motivations and tradeoffs between our method and others and their potential impact on performance and quality.

5.1 Test Setup

All results are obtained from a system with an i5-2400 CPU, 8GB of DDR3 RAM, and an NVIDIA GeForce GTX 970 GPU. The system is running Arch Linux kernel 4.16.8-1 and uses the proprietary NVIDIA graphics driver version 396.24. The application uses an OpenGL 4.5 context and the scene is rendered with a window resolution of 1920x1080.

The model used in the test scenes is the Crytek Sponza scene[1]. The textures

[1]The original Sponza model can be acquired from Crytek [35]. The model we use is modified by Alexandre Pestana [38].

from the original Sponza scene are replaced with ones necessary for physically-based rendering (materials are defined by a diffuse color, roughness, metallic coefficient, normal, and optionally an alpha texture). Mipmaps for the original textures were also precomputed and stored along with the full size 1024x1024 textures as DDS (Microsoft DirectDraw Surface) files using the compressed DXT5 format for faster scene loading.

To measure performance timings for each render pass we make use of OpenGL timer queries using the `GL_TIME_ELAPSED` query type. The results of the query object are double buffered to ensure introducing the timer query does not affect the total render time.

Unless otherwise mentioned, voxel resolutions are 256x256x256 with 6 mipmap levels.

5.2 Analysis

In the following sections we evaluate various aspects of our work. First, we discuss the global illumination algorithm as a whole. Second, we compare the rasterization-based and tessellation-based voxelization algorithms. Lastly, we provide the results of our voxel warping and some insights on future improvements.

5.2.1 Global Illumination

We test the algorithm with different screen resolutions and voxel grid resolutions. The rendered images with different voxel grid resolutions are shown in Figure 5.1. We see that the main difference with different voxel grid resolutions is how far the light spreads due to the larger voxels.

Table 5.1 shows timing results from the complete global illumination algorithm and Figure 5.2 plots the data as a stacked bar chart. Recall that a minimum goal

for real-time is 30 frames per second and a target is 60 frames per second, which correspond to individual frame times of 33.3ms and 16.7ms, respectively. We see that the total frame time always meets the goal of 60 frames per second.

The data shows that shadowmap creation and radiance injection was independent of screen resolution and voxel resolution[2]. The voxelization and radiance filtering steps both only depended on voxel grid resolution. The depth prepass also only varied with respect to screen resolution. The final shading predictably depended on both voxel grid resolution and screen resolution. However, larger voxel grid resolutions did not affect the final shading times by much at a given screen resolution, with approximately a 1.5ms difference between using a voxel resolution of 64 versus 256. Ultimately, the time spent for the shading dominates all other render passes and thus makes itself a prime target for any future work on optimizing for speed.

5.2.2 Tessellated Voxelization

The result of shading the scene with both voxelization methods is shown in Figure 5.3. First, we notice the rasterized approach generates 'smoother' voxels. This is a result of the rasterization discretizing the fragments as well as possibly not generating fragments for small triangles. The tessellated approach, since it uses the raw vertex positions to compute the voxel position, does not exhibit this smoothing effect. Also, with the rasterized approach we see cracks on the archs from imperfect conservative rasterization. The tessellated approach does not have issues with conservative rasterization. Instead, the issue is with large triangles (such as the floor): the GPU has a maximum supported tessellation level. Triangles that require a higher tessellation level than this hardware maximum will end up having holes[3].

[2] As expected, since the main factor for both of these render passes is the size of the shadowmap, which remained constant at 4096x4096.

[3] A possible workaround for this would be to subdivide large triangles before voxelizing, such as when loading in the mesh.

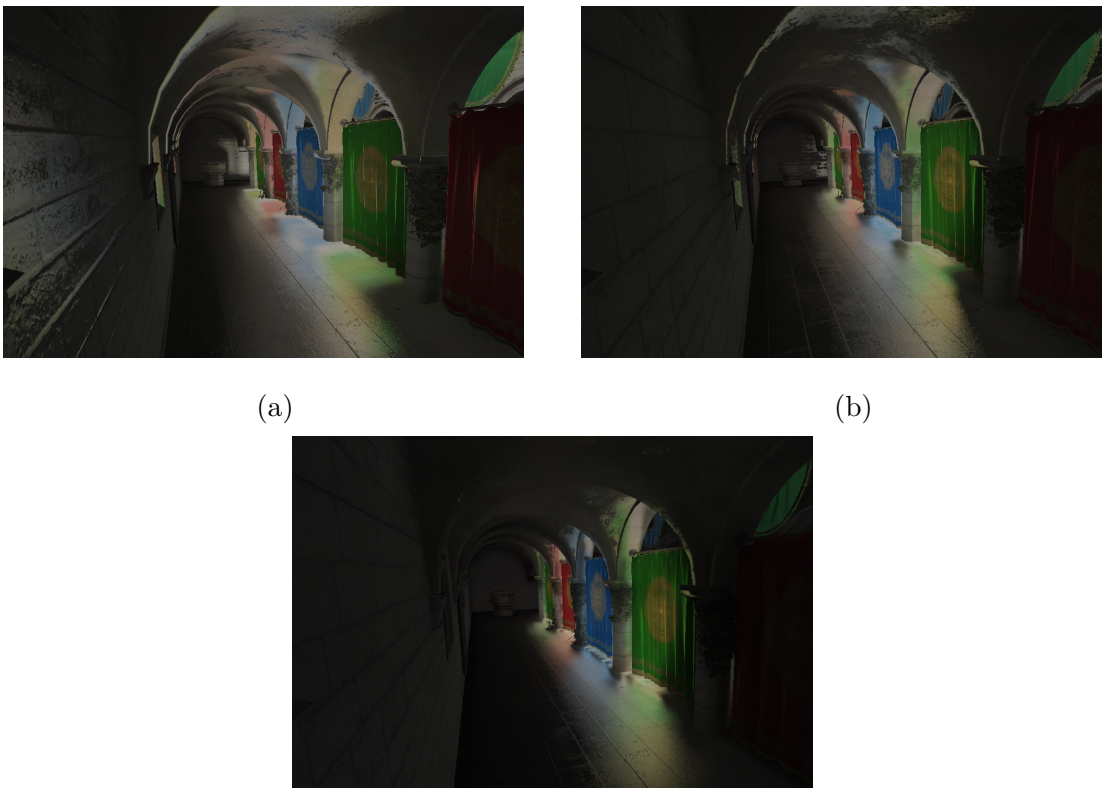


Figure 5.1: The scene rendered with voxel grid resolutions of 64^3, 128^3, and 256^3.

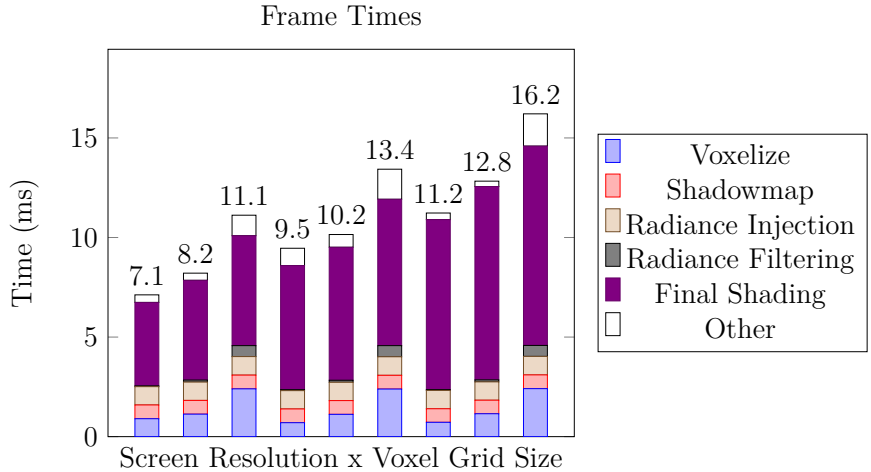


Figure 5.2: Graph showing the render pass times. Each bar shows the relative time contributions of each render pass. The depth prepass is incorporated into the Final Shading category and the Other category accounts for any other miscellaneous tasks done while rendering the frame.

Render Pass	Render Pass Time (ms)								
	1280x720			1600x900			1920x1080		
	64	128	256	64	128	256	64	128	256
Voxelize	0.90	1.13	2.40	0.70	1.12	2.39	0.72	1.15	2.41
Shadowmap	0.69	0.69	0.69	0.69	0.69	0.69	0.68	0.68	0.69
Radiance Injection	0.92	0.92	0.93	0.92	0.92	0.93	0.92	0.92	0.93
Radiance Filtering	0.04	0.10	0.55	0.05	0.10	0.56	0.04	0.10	0.55
Depth Prepass	0.20	0.20	0.20	0.25	0.26	0.25	0.31	0.31	0.36
Final Shading	3.89	4.81	5.32	5.98	6.42	7.10	8.23	9.40	9.75
Total	7.02	8.21	11.12	9.46	10.15	13.43	11.23	12.83	16.31

Table 5.1: Times measured for each render pass for various screen resolutions and voxel grid resolutions. The total timer also accounts for any other operations performed within each frame (i.e. the sum of the render pass times is not necessarily the complete time for an entire frame). Voxelization is done using the tessellation-based approach.

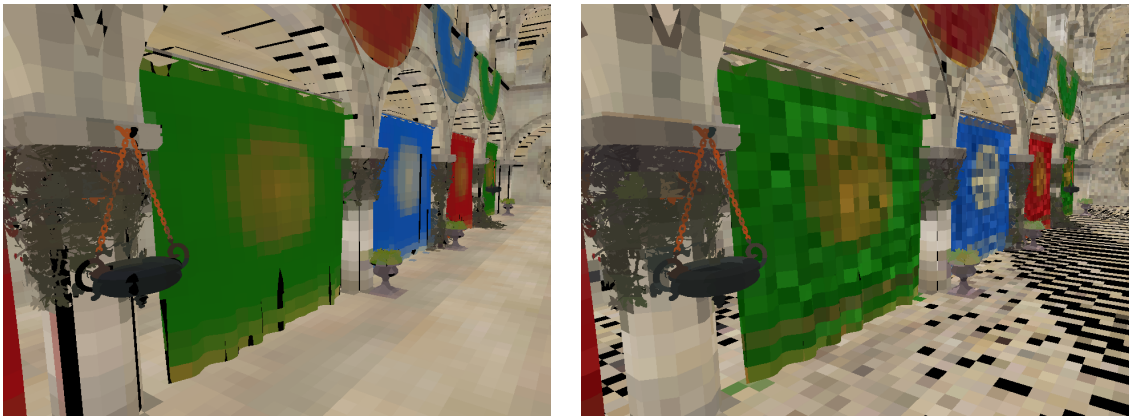
In terms of both performance and visual quality both voxelization methods are very similar. Voxelization times for each method are shown in Table 5.2 and graphed in Figure 5.6, where we see the tessellation-based voxelization is slightly slower than the rasterization-based approach[4]. Figure 5.5 compares the final rendered scene with both voxelization methods. The differences are minor.

Another limitation specific to the rasterized approach is shown in Figure 5.4. The fragment resolution must be set appropriately for the voxel density[5]. The tessellated voxels do not suffer from this issue since we do not rely on the rasterizer to produce fragments.

Ultimately, both voxelization methods are sufficient for real-time global illumination.

[4]However both implementations were not heavily optimized.

[5]This is especially important for the warped voxelization approaches, since the density is not uniform



(a) Rasterized voxels (b) Tessellated voxels

Figure 5.3: Comparison between the scene shaded based on the rasterized voxels (left) and the tessellated voxels (right).

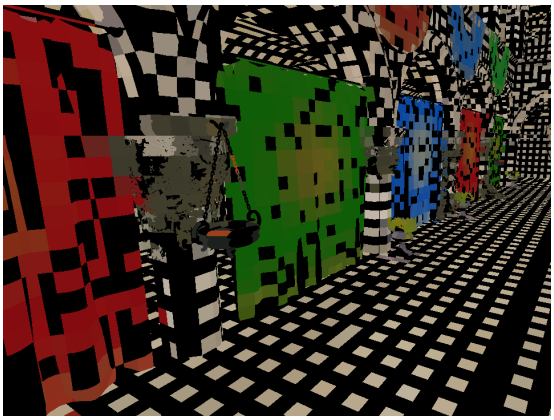


Figure 5.4: Image showing a limitation of the rasterized approach: the fragment resolution must be large enough for a particular voxel density. Otherwise, cracks will occur in the final voxelization.

The (unoptimized) tessellation-based approach is slightly easier to implement and debug[6] but is slower than the rasterization-based approach.

[6]Since the voxels are written in the tessellation evaluation stage, it is simple to add a geometry shader that takes the vertices and expands them into cubes, which are then rasterized and shaded with the vertex color (the same color inserted into the voxel texture).

(a) Rasterized voxels (b) Tessellated voxels

Figure 5.5: The final rendered image for both voxelization methods have negligible visual differences.

Voxel Grid Size	Voxelization Time (ms)	
	Rasterization-Based	Tessellation-Based
64x64x64	0.53	0.70
128x128x128	0.85	1.12
256x256x256	1.91	2.35

Table 5.2: Time spent voxelizing the scene with varying voxel grid resolutions. For the rasterization-based approach the MSAA method of conservative rasterization is used.

5.2.3 Integration of Voxel Cone Tracing into Existing Engines

Voxel cone tracing is an attractive method for adding full global illumination to existing engines. The necessary information needed for the algorithm should already be available in most engines. A voxelized representation of the scene can be generated from any arbitrary triangle mesh using either of the voxelization methods presented. Other methods for voxelization could also be used if, for example, some geometry is generated procedurally. For radiance injection the only external inputs required are shadowmaps (or RSMs) for any lights that will contribute to the indirect lighting.

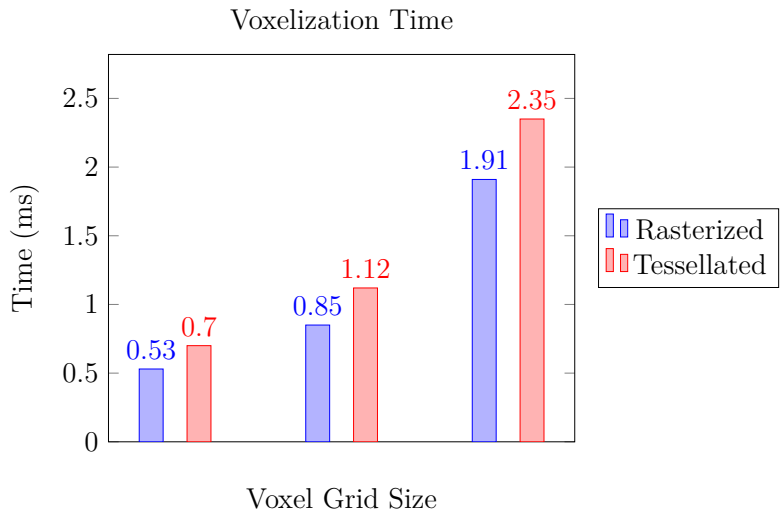


Figure 5.6: Graph comparing the voxelization time for both approaches at different voxel grid sizes.

Finally, the cone tracing needs surface normals and a TBN matrix (to transform the cone directions to world space), which will already be available for any engine which supports normal mapping.

5.2.4 Voxel Warping

Recall that the goal of voxel warping is to achieve a nonuniform voxel density. Ideally, this results in finer lighting detail for those areas that have increased density. Other approaches to increasing voxel resolution include clipmaps and octrees, as discussed in the Related Works. Fundamentally, however, the voxel resolutions were restricted to fixed sizes. With voxel warping we investigated how lifting this restriction would affect the voxelization quality.

The first method of voxel warping used a warp function to adjust the voxel density based on distance from the camera. Figure 5.7 shows a visual representation of the density. The effect of the warping on the final lighting is fairly minimal, as seen in Figure 5.8. The big problem with voxel warping arises when the camera moves, since the continuously changing resolution causes voxels to flicker as they move throughout

the voxel grid. Typically when the voxel sizes are all uniform this is fixed by snapping the voxel grid to discrete steps; but, with voxel warping, there is no single step size appropriate for all voxels.

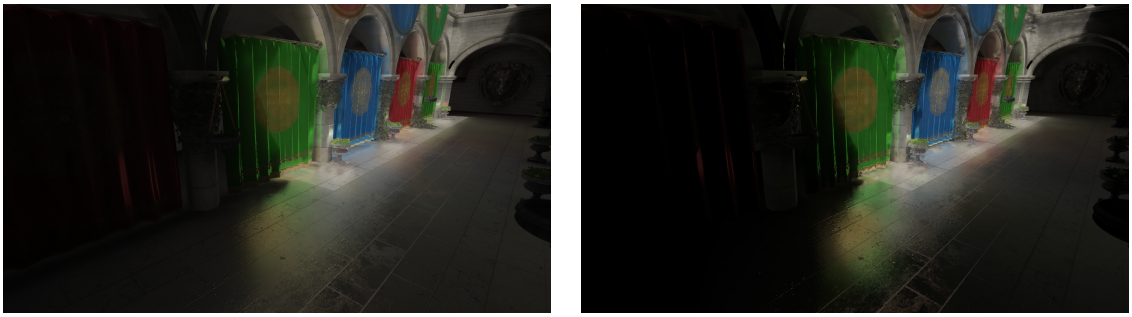
Figure 5.7: The scene is colored based on its density along the x axis (derived from the gradient of the warp function). Blue tints correspond to a slope of 2 and green corresponds to a slope of 1 (same as no warping).

(a) Without voxel warping

(b) With voxel warping

Figure 5.8: The voxel warping only has a small effect on lighting quality (the reflection of the green curtain is slightly more detailed).

The second method of voxelization utilizes perspective projection to determine the voxel sizes. The motivation behind this is for voxel sizes to correspond to their respective sizes in screen space. The lighting quality for this approach is noticeably more detailed than with the warp function, as seen in Figure 5.9, however the temporal flickering issues are still apparent. With future work, we think this can be alleviated or eliminated.



(a) Without voxel warping

(b) With voxel warping

Figure 5.9: The perspective voxel warping has a noticeable effect on lighting quality.

Chapter 6

CONCLUSIONS

In this work we implemented a real-time global illumination algorithm based on voxel cone tracing. We experimented with a tessellation-based voxelization method and nonuniform voxelization. Various aspects of our algorithm, including performance and visual quality, are evaluated and discussed. We find both rasterization-based and tessellation-based voxelization are similar in terms of both performance and voxelization quality. The voxel warping techniques provide promising results for further research. Most importantly, we provide a simple and cross platform implementation for future work.

6.1 Future Work

The goal of voxel warping was to increase voxel density for more detailed lighting as well as more efficiently use space within a 3D texture (and without the complexities of a sparse voxel octree). Unfortunately, having continuous voxel sizes makes minimizing temporal flickering difficult. To achieve the same goals while still having fixed voxel sizes, an approach to storing the voxels based on cascaded sparse 3D textures seems promising. Sparse textures (provided via the `ARB_sparse_texture` extension for OpenGL) are analogous to classic virtual memory systems: not all parts of the texture are actually allocated in memory. Only the parts of the voxel texture that are used would require memory (of course the implementation allocates in fixed-sized chunks, similar to pages in virtual memory). This approach would (ideally) combine the adaptive storage of octrees with the adaptive detail and hardware texture filtering of cascaded 3D textures.

For continued work on perspective voxel warping, exploring how to reduce temporal flickering would be worthwhile. Currently, voxels are aligned perpendicularly to the camera (a result of the view frustum having perpendicular near and far planes). Instead, curving the voxels may help reduce flickering when rotating the camera since the boundaries between adjacent voxels would be smooth. A possible way to do this might be to represent the voxels using spherical coordinates instead of cartesian coordinates.

Tessellated voxelization may lend itself well to a single-pass multi-resolution voxelization. Consider voxelizing a scene when using a cascaded 3D texture. With rasterization-based voxelization, the fragment resolution must be set to the maximum density required (for a single pass approach). This is wasteful for all areas of the texture where that high density isn't required. A multi-pass approach could be used instead, but this introduces overhead based on how many passes are needed. With tessellation-based voxelization, the density is configured dynamically in the tessellation control shader, and thus a multi-resolution voxelization should be achievable in a single pass without 'over-voxelizing' the lower resolution parts of the scene.

Another potential area of improvement is using alternative methods for storing radiance. For example, a spherical harmonics representation or ambient dice [19] representation could be used. This would have impacts on both lighting quality, performance, and memory usage. In addition, the use of anisotropic methods to reduce light filtering through solid objects could also be added.

An interesting topic to explore is dynamically adjusting the cone tracing step based on factors such as local geometric surface complexity or distance from the camera. For example, Panteleev discusses computing indirect lighting at a reduced resolution and then interpolating the resulting lighting for smooth surfaces [36]. Interpolation was also used for RSMs to reduce the amount of computation [14]. Other miscellaneous

optimizations for voxel cone tracing could also be pursued, such as only updating subregions of the voxel volume or filtered radiance at a time [33].

BIBLIOGRAPHY

[1] Mac computers that use OpenCL and OpenGL graphics. `https://support.apple.com/en-us/HT202823`. Accessed: 2018-05-11.

[2] NVIDIA VXGI. `https://developer.nvidia.com/vxgi`. Accessed: 2017-10-19.

[3] Tessellation — OpenGL Wiki, 2017. [Online; accessed 26-May-2018].

[4] T. Akenine-Möller and T. Aila. Conservative and tiled rasterization using a modified triangle set-up. *Journal of graphics tools*, 10(3):1–8, 2005.

[5] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[6] J. F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3):286–292, Aug. 1978.

[7] M. Bunnell. Dynamic ambient occlusion and indirect lighting. *Gpu gems*, 2(2):223–233, 2005.

[8] B. Burley and W. D. A. Studios. Physically-based shading at disney. In *ACM SIGGRAPH*, volume 2012, pages 1–7, 2012.

[9] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, Jan. 1982.

[10] C. Crassin. In *SIGGRAPH 2012 Course : Beyond Programmable Shading*. ACM SIGGRAPH, 2012.

[11] C. Crassin and S. Green. Octree-based sparse voxelization using the gpu hardware rasterizer. In P. Cozzi and C. Riccio, editors, *OpenGL Insights*, pages 303–319. CRC Press, July 2012.

[12] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 15–22, New York, NY, USA, 2009. ACM.

[13] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, volume 30, pages 1921–1930. Wiley Online Library, 2011.

[14] C. Dachsbacher and M. Stamminger. Reflective shadow maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pages 203–231, New York, NY, USA, 2005. ACM.

[15] J. de Vries. LearnOpenGL. Licensed under CC BY 4.0 by https://twitter.com/JoeyDeVriez.

[16] H. Doghramachi. Rasterized voxel-based dynamic global illumination. In W. Engel, editor, *GPU Pro 4*, pages 155–171. CRC Press, 2013.

[17] E. Eisemann and X. Décoret. Single-pass gpu solid voxelization for real-time applications. In *Proceedings of graphics interface 2008*, pages 73–80. Canadian Information Processing Society, 2008.

[18] Y. Fei, B. Wang, and J. Chen. Point-tessellated voxelization. In *Proceedings of Graphics Interface 2012*, GI '12, pages 9–18, Toronto, Ont., Canada, Canada, 2012. Canadian Information Processing Society.

[19] M. Iwanicki and P.-P. Sloan. Ambient Dice. In M. Zwicker and P. Sander, editors, *Eurographics Symposium on Rendering - Experimental Ideas & Implementations*. The Eurographics Association, 2017.

[20] Juan Linietsky, Ariel Manzur, and contributors. Godot engine.

[21] J. T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, Aug. 1986.

[22] V. Kämpe, E. Sintorn, and U. Assarsson. High resolution sparse voxel dags. *ACM Transactions on Graphics (TOG)*, 32(4):101, 2013.

[23] A. Kaplanyan. Light propagation volumes in cryengine 3. *ACM SIGGRAPH Courses*, 7:2, 2009.

[24] A. Kaplanyan and C. Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 99–107, New York, NY, USA, 2010. ACM.

[25] B. Karis and E. Games. Real shading in unreal engine 4. *Proc. Physically Based Shading Theory Practice*, 2013.

[26] A. Keller. Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[27] J. Kessenich. *The OpenGL Shading Language*. The Khronos Group Inc., June 2017. Version 4.5.

[28] S. Lagarde and C. Rousiers. Moving frostbite to physically based rendering. *part of ACM SIGGRAPH2014 Course: Physically Based Shading in Theory and Practice*, 2014.

[29] S. Laine and T. Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, 2011.

[30] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, Aug. 1987.

[31] F. Losasso and H. Hoppe. Geometry clipmaps: Terrain rendering using nested regular grids. *ACM Trans. Graph.*, 23(3):769–776, Aug. 2004.

[32] M. McGuire, M. Mara, D. Nowrouzezahrai, and D. Luebke. Real-time global illumination using precomputed light field probes. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '17, pages 2:1–2:11, New York, NY, USA, 2017. ACM.

[33] J. McLaren. Graphics deep dive: Cascaded voxel cone tracing in the tomorrow children. `https://www.gamasutra.com/view/news/286023/Graphics_Deep_Dive_Cascaded_voxel_cone_tracing_in_The_Tomorrow_Children.php`, 2016. Accessed: 2017-10-19.

[34] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129 – 147, 1982.

[35] F. Meinl. Sponza atrium. `http://www.crytek.com/cryengine/cryengine3/downloads/`.

[36] A. Panteleev. Practical real-time voxel-based global illumination for current gpus. `http://on-demand.gputechconf.com/gtc/2014/presentations/S4552-rt-voxel-based-global-illumination-gpus.pdf`, 2014. Accessed: 2018-05-02.

[37] M. Paulin. *GigaVoxels: A voxel-based rendering pipeline for efficient exploration of large and detailed scenes*. PhD thesis, Karlsruhe Institute of Technology, 2011.

[38] A. Pestana. Base color, roughness and metallic textures for sponza. `http://www.alexandre-pestana.com/pbr-textures-sponza/`.

[39] M. Pharr, W. Jakob, and G. Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[40] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[41] M. Schwarz and H.-P. Seidel. Fast parallel surface and solid voxelization on gpus. *ACM Trans. Graph.*, 29(6):179:1–179:10, Dec. 2010.

[42] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification*. The Khronos Group Inc., June 2017. Version 4.5.

[43] P.-P. Sloan, J. Kautz, and J. Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Trans. Graph.*, 21(3):527–536, July 2002.

[44] J. Story. Don't be conservative with conservative rasterization. `https://developer.nvidia.com/content/dont-be-conservative-conservative-rasterization`, November 2014.

[45] M. Takeshige. Basics of GPU voxelization. `https://developer.nvidia.com/content/basics-gpu-voxelization`, March 2015.

[46] C. C. Tanner, C. J. Migdal, and M. T. Jones. The clipmap: A virtual mipmap. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 151–158, New York, NY, USA, 1998. ACM.

[47] WhiteTimberWolf. Octree2. `https://commons.wikimedia.org/wiki/File:Octree2.svg`, 2010. Licensed under CC BY-SA 3.0 by WhiteTimberWolf.

[48] L. Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, Aug. 1978.