REAL-TIME RAY TRACED GLOBAL ILLUMINATION USING FAST SPHERE INTERSECTION APPROXIMATION FOR DYNAMIC OBJECTS

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

Reed Garmsen

February 2019
COMMITTEE MEMBERSHIP

TITLE: Real-Time Ray Traced Global Illumination using Fast Sphere Intersection Approximation for Dynamic Objects

AUTHOR: Reed Garmsen

DATE SUBMITTED: February 2019

COMMITTEE CHAIR: Christian Eckhardt, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Zoë J. Wood, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Franz J. Kurfess, Ph.D.
Professor of Computer Science
Realistic lighting models are an important component of modern computer generated, interactive 3D applications. One of the more difficult to emulate aspects of real-world lighting is the concept of indirect lighting, often referred to as global illumination in computer graphics. Balancing speed and accuracy requires carefully considered trade-offs to achieve plausible results and acceptable framerates.

We present a novel technique of supporting global illumination within the constraints of the new DirectX Raytracing (DXR) API used with DirectX 12. By pre-computing spherical textures to approximate the diffuse color of dynamic objects, we build a smaller set of approximate geometry used for second bounce lighting calculations for diffuse light rays. This speeds up both the necessary intersection tests and the amount of geometry that needs to be updated within the GPU’s acceleration structure.

Our results show that our approach for diffuse bounced light is faster than using the conservative mesh for triangle-ray intersection in some cases. Since we are using this technique for diffuse bounced light the lower resolution of the spheres is close to the quality of traditional raytracing techniques for most materials.
ACKNOWLEDGMENTS

Thanks to:

- My parents, for their endless love and support throughout the good, the bad, and the in between
- My friends, especially Matthew Callaghan, Calin Washington, Mason Wright, and Ken Yasui, without whom I would have lost my motivation long ago
- Professor Christian Eckhardt and Professor Zoë Wood, for their consistent help navigating the world of computer graphics
- The many professors at Cal Poly, who supported me throughout my journey here
- Andrew Guenther, for uploading this template
# TABLE OF CONTENTS

| LIST OF TABLES                                    | viii |
| LIST OF FIGURES                                  | ix  |
| CHAPTER                                          |     |
| 1 Introduction                                   | 1   |
| 1.1 Global-Illumination                         | 2   |
| 1.2 Real-Time Raytracing                         | 3   |
| 1.3 Our Contribution                             | 4   |
| 2 Background                                     | 5   |
| 2.1 The Graphics Pipeline                        | 5   |
| 2.1.1 Raytracing                                 | 5   |
| 2.1.2 3D Data Structures                         | 10  |
| 2.2 The DirectX 12 API                            | 12  |
| 2.3 DirectX Raytracing                            | 14  |
| 2.3.1 Detailed Overview                          | 15  |
| 2.3.2 Shaders                                    | 18  |
| 2.3.3 Acceleration Structure                     | 20  |
| 3 Related Works                                  | 24  |
| 3.1 Monte Carlo Ray Tracing                       | 24  |
| 3.2 Voxel Cone Tracing                            | 26  |
| 4 Implementation                                 | 29  |
| 4.1 Interacting with DirectX and DXR              | 29  |
| 4.1.1 DirectX Raytracing pipeline                 | 29  |
| 4.1.2 Uploading Geometric and Texture Resources   | 31  |
| 4.1.3 Creating the Shader Table                   | 31  |
| 4.2 Raytracing                                    | 32  |
| 4.3 Monte Carlo Global Illumination with DXR      | 33  |
| 4.4 Diffuse Spherical Approximation               | 34  |
| 4.5 Note on Denoising                            | 36  |
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>The polygon counts for each triangle mesh used in the project. Spherical geometry is not included as they are not triangle meshes.</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>Frame rates (frames per second) when only the Sponza scene is included averaged over 5 seconds (all FPS counts are averaged over 5 seconds, with FPS polled once a second). Times marked N/A failed to run due to a GPU hang (TDR, or related crash). Rendered frame is shown in Figure 5.1.</td>
<td>39</td>
</tr>
<tr>
<td>5.3</td>
<td>This table measures FPS with the Sponza scene and various amounts of the &quot;Old Man&quot; character mesh. Resolution is held constant at 640 x 480. The sphere approximation method is not used for the GI measurements.</td>
<td>40</td>
</tr>
<tr>
<td>5.4</td>
<td>This table measures FPS with conditions equivalent to Table 5.3. However, it uses the sphere approximation method for GI measurements as an exception.</td>
<td>43</td>
</tr>
<tr>
<td>5.5</td>
<td>This table measures FPS with conditions equivalent to Table 5.3. However, we’ve removed most of the Closest Hit shader logic from secondary bounces to try to narrow performance differences to the intersection logic.</td>
<td>46</td>
</tr>
<tr>
<td>5.6</td>
<td>This table measures FPS with conditions equivalent to Table 5.4. However, we’ve removed most of the Closest Hit shader logic from secondary bounces to try to narrow performance differences to the intersection logic.</td>
<td>47</td>
</tr>
<tr>
<td>5.7</td>
<td>This table measures FPS with conditions equivalent to Table 5.5. However, it uses the higher resolution old man model described in Table 5.1.</td>
<td>48</td>
</tr>
<tr>
<td>5.8</td>
<td>This table measures FPS with conditions equivalent to Table 5.6. However, it uses the higher resolution old man model described in Table 5.1.</td>
<td>49</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An example of color bleeding. Here the curtains are bleeding their color onto the floor of the Sponza scene.</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>A diagram of the raytracing algorithm. Rays are traced from the virtual camera through the image which then either intersect or do not intersect objects. Shadows rays are also cast to determine light visibility from the initial point of intersection. CC BY 4.0 [9].</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>An example of a raytraced image with a reflective sphere in the center and two refractive glass spheres on either side. A depth of field effect is also present blurring objects in the foreground. Reflection and refraction are natural to implement in a raytracer without any noticeable compromises.</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Diagram of a basic octree. Each non-leaf node has eight children that contain some subset of the scene (denser areas are subdivided further). Leaf nodes would typically contain a single piece of geometry. CC BY-SA 3.0 [29].</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Diagram of a 2D bounding volume hierarchy, or BVH. Here the B and C blocks were subdivided along the X-axis with B being subdivided again along the X-axis. This leads to an even distribution of the objects in this scene. CC BY-SA 3.0 [20].</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>An example of a DXR shader table. A hit group shader record contains everything the DXR pipeline will need when it intersects a piece of geometry (and by extension then offsets to the correct shader record in the table). Everything in each shader record is initialized prior to calling DispatchRays(). CC BY-NC-ND 4.0 [30].</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>The steps performs once DispatchRays() has been called. While each stage is discussed in detail, this gives an overview on how they fit together.</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>A visual representation of Kajiya’s rendering equation. A portion of all incoming light over the hemisphere is reflected along the vector to the camera/eye. Monte Carlo global illumination is concerned with how we can approximate adding up all the incoming light at point x, before we determine how much is reflected to the camera. CC BY-SA 3.0 [25].</td>
<td>25</td>
</tr>
</tbody>
</table>
3.2 An example render from the original voxel cone tracing application by Crassin, et al [3]. Note the full specular and diffuse reflections from the curtains in the Sponza scene. 

4.1 The left picture shows a normal boot triangle mesh. The left picture is that same boot converted to a spherical texture representation and applied to a sphere. The sphere is necessarily larger than the boot in areas due to the boot not being perfectly round.

5.1 A 1920 x 1080 resolution render of the Sponza scene using our application. No dynamic geometry is included but GI is enabled with 32 samples and a single GI bounce. This shot of our scene is what we used for all our benchmarks unless otherwise noted.

5.2 Graph of the data from Table 5.3.

5.3 An example image of adding the "Old Men" as dynamic objects to the scene. There are 10 men evenly spaced here. When the number of men is increased, the length covered remains the same but the density of objects increases.

5.4 The effect of enabling diffuse global illumination is immediately apparent between the images with and without it. Increasing the sample count to from 8 to 32 begins to clean up the image noise (though it is still noticeable).

5.5 Graph of the data from Table 5.4.

5.6 Graph of the change in FPS values from Table 5.3 to 5.4. All comparison charts leave off data points where zero change occurred (i.e. when no GI samples are used), or when one or both tables have incomplete data (fields marked 'N/A' or '-').

5.7 Graph of the data from Table 5.5.

5.8 Graph of the data from Table 5.6.

5.9 Graph of the change in FPS values from Table 5.5 to 5.6.

5.10 Graph of the data from Table 5.7.

5.11 Graph of the data from Table 5.8.

5.12 Graph of the change in FPS values from Table 5.7 to 5.8.
5.13 Here we compare the result of using spherical approximation of the boot with the full mesh model for GI. Notably it appears that the orange accent along the top of the boot is over-represented in the spherical model causing too much bounced orange light at the top of the pillar (the upper pink area). On the other hand, the concave toe of the boot fails to bounce significant orange light on the lower portion of the column (the lower pink area). The downside of only using one bounce of global illumination is that some areas remain completely dark.

5.14 Following from the previous comparison in Figure 5.13, we again see the orange trim being over-represented along the top of the boot. Overall, this angle gets better results than the previous comparison as the back of the boot is not concave, which caused issues in the previous image. We also see minor differences along the boot due since it won’t receive ”self-inflicted” bounced light contributions in the approximation like in the full mesh.

5.15 More comparisons following Figures 5.13 and 5.14. Here the boot is placed in the corner of two white walls with the light shining perpendicular to the boot. Noise is much more pronounced due to the simple background, but it is easier to see the differences described in the previous figures.

5.16 A similar shot to Figure 5.15 but all lighting has been removed except for diffuse GI from the boot to maximize visibility.
Chapter 1

INTRODUCTION

Computer graphics is one of the most computational and data-intensive fields of computer science. Whether it’s needing to throughput more triangles to the GPU to increase the realism and complexity of models or running complicated lighting equations to try to simulate different lighting conditions, computer graphics applications always need more hardware power and software optimization. However, until we have a computer with near infinite power and storage, compromises and approximations will always be needed to be made to render our world as accurately as possible.

To that end, lighting computation is one of the more expensive steps when rendering a scene depending on how accurate the simulation is. Offline rendering (rendering where time to complete is not critical) often spends the majority of run time computing the lighting for each individual pixel in the scene, for a single rendered frame. This can take hours depending on the complexity of the scene being rendered. While this is acceptable for some applications such as animated movies, real-time applications (where time to render a frame can be as short as 16.67ms, or 60 frames per second) does not have this luxury. Often, trade-offs of quality are made to meet real-time performance and speed targets.

Our contribution attempts to help bridge the gap between offline and real-time rendering algorithms for fast yet accurate lighting computation. While many aspects go into the rendering pipeline, we focus on the bounced or indirect light of a scene that often challenges typical real-time pipelines.
1.1 Global-Illumination

Lighting computations are typically broken up into two groups, direct lighting and indirect lighting. Direct lighting is further broken up into diffuse lighting (lighting affected by the micro-facets of the surface being reflected) and specular lighting (lighting due to the reflection of a light source, easily seen on a glossy surface). Calculating direct lighting for a scene requires less computation than indirect light as it is primarily dependent on the number of lights in the scene.

Indirect lighting is any lighting that doesn’t come directly from a light source. A classic example is color-bleeding. This is where an object is directly lit and that object then reflects some light onto nearby objects. Those objects are then subtly lit by color that is “bled” from the object that was directly lit, effectively making every object in the scene a possible light source. This effect is subtle, but greatly adds to the realism of a scene and is why objects in shadow are not pitch black in reality. Indirect lighting is much slower to compute and thus accuracy is still an issue when rendering in real-time.

This level of illumination is hard to compute when using triangle rasterization. This is because of how many objects must be considered as "lights" in the scene. Instead of a scene having a discretized number of objects that light the scene (e.g. the sun and some light bulbs), the entire scene can potentially bounce and/or emit light. This has necessitated the use of heavy approximations, such as simply setting indirectly lit pixels to a default, "ambient" value based on the color of lights in the scene. While fast, these approximations do not have the quality necessary for truly realistic scenes.
Figure 1.1: An example of color bleeding. Here the curtains are bleeding their color onto the floor of the Sponza scene.

1.2 Real-Time Raytracing

Global illumination algorithms used within the framework of traditional rasterization often struggle to compute accurate global illumination without pre-computation or additional techniques like Voxel Cone Tracing [3] (which borrows heavily from raytracing). Comparatively, the raytracing algorithm is designed around how light is bounced around the scene (see Section 2 for more details on raytracing). Occasionally, raytracing concepts will been used in real-time applications in effects such as Screen-Space Reflections (SSR) [15] and Voxel Cone Tracing [3]. However, recent developments in the GPU software and hardware space have started to change this, and real-time raytracing is becoming more feasible with commodity hardware [19].

Real-time raytracing is desired by graphics developers for both simplifying the graphics pipeline and allowing for accurate and realistic lighting regarding things like
bounced light and translucent objects. Real-time raytracing is still very expensive and requires large amounts of software optimization to be feasible in real-time products, such as video games [19].

One issue unique to raytracing versus rasterization is needing to determine the specific piece of geometry (e.g. a triangle of a character mesh) that each ray intersects. With modern character models reaching multiple millions of triangles, even with advanced 3D data structures, this computation can be expensive and slow. The additional need to have dynamic objects requires that the 3D data structure have a mechanism to update itself which slows things down further (DirectX Raytracing’s approach is discussed in Section 2).

1.3 Our Contribution

Our contribution described in this work is thus an attempt to help speed up the intersection process and reduce the number of intersections needed by the DirectX Raytracing API (DXR) by approximating dynamic geometry with textured spheres. The advantage of DXR or a similar API is in using heavily optimized GPU driver code as well as new fixed-function hardware within the GPU that is designed to speed up raytracing. With further hardware and driver advancements (such as a built-in sphere intersection function), we think our solution would greatly improve the feasibility of using raytracing for second bounce global illumination in real-time applications.

We also think the following thesis presents a good investigation into DXR and where programmers new to the API may run into pitfalls. Working with an API that reached version 1.0 in October of 2018 [28] presents many challenges that are worth discussing. We hope that this work will help programmers interested in DXR as well as software and hardware GPU developers looking to improve raytracing APIs.
2.1 The Graphics Pipeline

The graphics pipeline can be defined as the series of steps necessary to move from some sort of input (like triangle vertices and material values) to a set of colored pixels that make up a rendered image as the output. The exact steps required vary depending on specific features, but mainly on the primary rendering approach used. While others exist, the two dominant approaches are the rasterization and raytracing rendering algorithms. We discuss raytracing in depth below and a good introduction to rasterization can be found in [21].

2.1.1 Raytracing

While triangle rasterization is very fast when it comes to processing geometry, it struggles with more complicated lighting simulation as it doesn’t map naturally to how light works in reality. Raytracing closer maps to how light works in reality at a simple level. This allows programmers to more easily implement and maintain many more complicated lighting algorithms. The trade-off is a more complicated and expensive visibility test per pixel.

Before jumping into how raytracing works it helps to briefly discuss how light works in real life as a particle. Light is emitted from light sources (emitters) as packets of light, or photons. This light is transmitted and absorbed by objects or some sort of transmission media (e.g. fog) along it’s initial direction. Each bounce of the light contributes less energy as it loses energy to absorption and re-emission [18].

The number and direction of photons expelled from the emitter for a light source
Figure 2.1: A diagram of the raytracing algorithm. Rays are traced from the virtual camera through the image which then either intersect or do not intersect objects. Shadows rays are also cast to determine light visibility from the initial point of intersection. CC BY 4.0 [9].

like the sun would make it hard to accurately simulate in a reasonable amount of time for a real-time application (approximations like photon mapping [11] exist, but still use ray tracing for accumulation). Because of that, ray tracing inverts the problem and instead traces a ray starting from the camera/eye position. One ray is traced through each pixel to determine the color of that pixel by intersecting with the nearest piece of geometry. Once an intersection is found, the color at that location is calculated and the pixel shaded. If no intersection is found, the programmer can default to a single color or index the ray into some sort of background image (typically known as a skybox).
Geometry Intersection

As previously mentioned, before we can calculate a pixel’s color we need to determine the geometry visible from that pixel (we need information such as an object’s normal, and material properties). This requires that we perform an intersection test on the scene geometry with the ray. Many types of geometry can be intersected with rays and used in raytracing ([6] has links to intersection test implementations for many common geometry types). Two commonly used in raytracing are triangles and spheres. Triangles are at the foundation of almost every type of graphical and modeling software, making them necessary regardless of speed. Spheres are much less common in something like rasterization (where a sphere would just be a series of discrete triangles rather than a true mathematical sphere), but they are used in raytracing.

If the ray does intersect a piece of geometry, we then store the distance along the ray of the hit and continue intersecting geometry. If we find an intersection that happened earlier along the ray, we overwrite the previous depth value and associated information geometric information (used in the final lighting calculations).

Direct Lighting

In a basic ray tracer, all we need is the closest hit to determine lighting calculations which returns your traditional lighting information as described above. We can then use standard shading algorithms like Phong or Blinn-Phong to return the color of that pixel [1]. This step is practically identical to the rasterizer version.

A notable exception here is shadows. To see if the hit location is considered in shadow, we can trace an additional ray from the closest hit point to each light source. If there is no intersection from that ray to a piece of geometry (bounded by the
distance to the light), we calculate the lighting contribution from that light. If there is a hit, we don’t count that light’s contribution. If every ray to each light intersects geometry, the closest hit is considered in shadow. We then only use indirect lighting calculations to color that pixel.

**Indirect Lighting**

Indirect lighting is where the power of raytracing starts to come into play. As previously discussed, light can bounce around a scene quite a bit. This means that to calculate how light affects a closest hit (i.e. how we should shade that pixel) we need to cast more rays into the scene. Using the closest hit as the starting point of the ray, those additional rays are then used to determine incoming light that is not direct from a light source.

There are multiple ways of considering how and where to shoot out rays to check for indirect light contributions (see Monte Carlo raytracing in Section 3 for a detailed starting point). The idea is that you need to take shoot out a large number of rays from *each* closest hit. And those recursive indirect light rays can also shoot out their own sampling rays for increased accuracy to account for light bouncing multiple times.

Because of this, indirect lighting calculations quickly become very expensive as the number of sample rays increase and as the ray depth increases (one or two is typically enough). Therefore, a common goal is to reduce the number of rays needed for accurate lighting. This includes making the bounced rays used be more useful via a specific sampling pattern, or by relying on *denoising* techniques to get the number of samples down to numbers as low as one [13].
Figure 2.2: An example of a raytraced image with a reflective sphere in the center and two refractive glass spheres on either side. A depth of field effect is also present blurring objects in the foreground. Reflection and refraction are natural to implement in a raytracer without any noticeable compromises.

Other Lighting Considerations

Raytracing also has the benefit of naturally simulating lighting conditions caused by pure reflections and refraction. Consider a glass sphere. Light is going to both reflect off the surface as well as refract through the glass medium of the sphere and pass through the other side.

Refraction is modelled by using the index of refraction of glass with *Snell’s Law* to calculate the refracted ray through the sphere. Then it can then determine if there was another piece of geometry (and perform the requisite lighting calculation)
or handle the ray miss appropriately.

2.1.2 3D Data Structures

The key difference between raytracing and rasterization is a shift of focus from processing individual geometry one at a time (by triangle) to the image itself. There is now a need to find the "closest" piece of geometry for every ray in the scene (so at least width x height rays if only the primary, per-pixel rays are used). In its most naive form, this step requires performing an intersection test for every piece of geometry in the scene for every ray. If the scene only consists of 10 triangles and a few spheres, this is relatively quick. However, the scene of the typical modern video game has many more triangles (a single character in the 2013 game *Killzone: Shadow Fall* hits 40,000 triangles alone [27]) and performing an intersection test on each triangle isn’t a trivial operation itself as seen in the various triangle intersection tests listed in [6].

To get around needing to intersect every piece of geometry, the common approach is to wrap groups of geometry in larger geometric primitives. For example, with 10 triangles, we could wrap two groups of 5 triangles within a sphere. Then, we need only intersect both spheres to determine the closer group. From there, only 5 triangles need to be intersected rather than the full 10. This kind of data structure is very common in graphics problems and are referred to collectively as 3D data structures since they break up 3D space. Below are two examples of common 3D data structures that are used in raytracing. Both are valid approaches as they adaptively subdivide themselves to best match the layout and composition of the scene, to different degrees.
Octrees

An octree is a tree data structure with uniform sized nodes (see Figure 2.3). The core of the idea is to subdivide the space of the scene into smaller and smaller chunks, starting with 8 (hence the "oct" in octree) smaller uniform boxes within a larger box that encompasses the entire scene. These smaller boxes are then subdivided further until a maximum threshold of objects per box is hit (e.g. 3 primitives per box) or a maximum tree depth is reached. Thus, the densest parts of the scene have the deepest nodes whereas relatively sparse areas of the scene may only have one or two layers of subdivision, if any.

This naturally maps to solving the raytracing intersection issue by subdividing the scene into chunks for more efficient traversing (the exact speed-up is dependent upon the density and distribution of the geometry of the scene).
Figure 2.4: Diagram of a 2D bounding volume hierarchy, or BVH. Here the B and C blocks were subdivided along the X-axis with B being subdivided again along the X-axis. This leads to an even distribution of the objects in this scene. CC BY-SA 3.0 [20].

Bounding Volume Hierarchy

A bounding volume hierarchy (BVH) is similar to an octree in that it is also a tree data structure that dynamically divides the scene into hierarchical groups of smaller and smaller chunks according to the density of the scene. However, a BVH does not build a uniform grid. Instead, a BVH will "cut" the current node (starting with a node encompassing the whole scene) along either the X, Y, or Z dimension, repeating until a threshold of objects per node is hit for the leaf nodes.

To jump ahead a bit, the exact acceleration structure used by the drivers backing the DXR API is implementation defined. However, its important to note that one is always used and will likely match or borrow from the ideas discussed here.

2.2 The DirectX 12 API

DirectX 12 is, as of this writing, the most recent version of Microsoft’s DirectX graphics API used for Windows. DirectX and APIs like it (OpenGL, Metal, and Vulkan, to name a few) are designed to allow users to interact with the GPU and facilitate graphics related requests (e.g. sending triangles vertices to the GPU to
be drawn to the screen with a specific set of shaders). While still fairly low-level (especially with DirectX 12, Vulkan, and Metal), these APIs abstract away driver and hardware vendor specific details and ensure a common interface for many different GPUs.

Traditional GPU APIs, such as OpenGL and DirectX versions before 12, took the approach of managing as much state required to interact with the GPU as possible [10]. This allowed programmers to focus on their own application rather than wrangling with GPU residency management and memory allocation techniques. However, this automatic management resulted in a slower API as the driver typically had to manage these things without knowledge of your specific application. They also had trouble allowing API calls from multiple CPU threads. As graphics programmers constantly want to eek out more performance, the idea of letting the programmer manually manage this driver-level state was considered. Two of the major APIs to support this idea are DirectX 12 and Vulkan.

These APIs are noticeably more complex (DirectX 12 requires “significant graphics expertise” [10]) compared to their more managed counterparts. The complication comes from many tasks requiring more code and thought that the application programmer is now responsible for. Where before a graphics programmer could use a texture resource however she wanted at any point, she now must have memory barriers set up to tell the driver exactly where, when, and how she plans to use that resource (before it would figure that out most of that for the programmer, at the cost of some overhead). Failing to correctly set-up a set of calls to the GPU often results in a dreaded ”TDR”, or ”Timeout Detection and Recovery” where the GPU effectively has a segmentation fault and stops responding, forcing a reset of the GPU [24]. This makes errors in GPU logic much harder to detect, without the help of programs such as Microsoft’s PIX [23], which allow you to do advanced analysis on a full GPU frame capture of your application.
Overall, an advanced API like DirectX 12 is worthwhile to professionals and advanced students. It is an expert-level API compared to those that existed before it and is not recommended as a starting point for those new to graphics. In the opinion of the authors, an exception might be a wrapper library for it that effectively makes it more similar to something like DirectX 11 (again handling much of the GPU stack and resource management automatically). This would then allow for new users to slowly peel back the complexity as they learn computer graphics fundamentals on the GPU.

On the other hand, DirectX Raytracing, the API extension for DirectX that takes advantage of driver and hardware support for raytracing on the GPU is only available as part of DirectX 12 [19]. So, for our contribution, we chose DirectX 12 as the API of choice by default.

2.3 DirectX Raytracing

DirectX Raytracing (DXR) was revealed and launched in 2018 [19]. By integrating directly with an existing API and having driver/hardware support, it has both easier integration with existing graphics applications and pipelines (especially those already using DirectX 12) and allows for hybrid rasterization/raytracing pipelines. While a few in-depth resources exist like the full functional specification for DXR (which can be viewed via the post here [4]), below we present the basics of the API necessary to understanding our contribution.

The key to understanding DXR is first acknowledging that its design is in some ways reversed from current API design trends. Originally, graphics APIs were designed to be fixed-function, meaning that instead of programmable shaders programmers were forced to embrace standardized API calls for doing things like fragment manipulation. This was done to take advantage of very specialized GPU hardware
that was far from general purpose. Now, GPUs are more general in design for parallel problems. They are often used for tasks outside of graphics, like Machine Learning inference and training. DirectX Raytracing keeps many things general purpose, like the concept of shaders (there are five new ones for DXR specifically), but leaves some things to be driver and hardware defined for performance reasons. Key among those is the acceleration structure used. We discuss both shaders and the acceleration structure below.

2.3.1 Detailed Overview

Being an extension of DirectX 12 and its rasterization pipeline, DXR is similar where possible to base DX12 but has significant differences from the rasterization path due to raytracing being a fundamentally different algorithm. A big difference is that, where traditional DX12 and rasterization is very triangle driven (you make draw calls on a set of vertices with specific settings for things like material and color), DXR is ray driven. Instead of calling draw on a set of triangle vertices, you simply call \textit{DispatchRays()}, which will shoot one ray per-pixel\(^1\) for the ray tracing algorithm. In addition, we need to have all scene information (geometry, material values, etc) ready in case any of the dispatched rays intersect with a given piece of geometry.

This mandates careful management and setup of your resources before even trying to draw anything. To accomplish this, DXR provides a number of subsystems that the programmer must use.

First, the Acceleration Structure (described in-depth below) provides the structure in which to load geometric information and pass it to the GPU. This allows fast traversal on the GPU via a standard, fixed-function interface for driver vendors to

\(^1\)By extension it can be an arbitrarily sized texture, but the standard is one ray for every pixel when you’re rendering to a screen. Techniques like anti-aliasing may call for more than one ray per-pixel as well.
Figure 2.5: An example of a DXR shader table. A hit group shader record contains everything the DXR pipeline will need when it intersects a piece of geometry (and by extension then offsets to the correct shader record in the table). Everything in each shader record is initialized prior to calling DispatchRays(). CC BY-NC-ND 4.0 [30].

Second, is the concept of the Shader Table. At a high level, these can be thought of as a bookkeeping table for all resources required to render the geometry in the scene, from the shaders used, to the textures and constant values that help define it.

Shader Tables are made up of individual records which contain the necessary information to render one piece of geometry (or a set of geometry if the shader record is the same, e.g. a set of triangles) as seen above in Figure 2.5. These records are indexed by the driver using a given size of the shader record (all shader records must be the same size, meaning size is determined by the largest record in the table [22]) and an internal indexing scheme as the ray traverses the acceleration structure.

Once all the necessary information is set-up and copied to the GPU, the application programmer is free to call DispatchRays(). From that point the ray tracing pipeline takes over, a diagram of which is shown in Figure 2.6.

We talk about each area in detail below using [22, 4] and our own experience as
Figure 2.6: The steps perform once `DispatchRays()` has been called. While each stage is discussed in detail, this gives an overview on how they fit together.
2.3.2 Shaders

Ray Generation

The *Ray Generation* shader is responsible for generating rays. The number of threads is set-up on the CPU and typically all result in a call to *TraceRay()* which performs a ray trace on a single ray given an initial direction vector and starting point (typically the camera is the starting point and the direction is from the camera through the virtual pixel in world space). Some additional parameters include maximum distance along the ray (which can be used to not render far away objects), and other miscellaneous flags.

The rays then determine object intersection and distance by polling the acceleration structure. Exactly what happens here depends on the shader choices of the application programmer.

Intersection

The *Intersection* shader performs an intersection test according to the geometric definition of the primitive. In other words, if the piece of geometry is a triangle, the intersection shader should perform the math necessary to test if the given ray (from the ray generation shader) intersects that triangle. If so, it also needs to return the distance along the ray that the intersection occurred (which is necessary to determine the closest object intersected along the ray if multiple such objects exist). The intersection shader for a triangle is built-in for DXR and can be selected by the application programmer (meaning no intersection shader needs to be written). Other geometric primitives, such as spheres and cubes, do not have this built-in functionality.
and therefore require custom intersection shaders.

**Any Hit**

The *Any Hit* shader was not used for our contribution. However, we will briefly describe its use and possible function for sake of completeness.

After an intersection test is performed and returns that there was a successful hit, the application programmer can *optionally* prescribe an additional shader step before moving on. This *Any Hit* shader can be used to gather information from things like partially transparent objects that you don’t want to count as the ”closest” object for sake of stopping the ray traversal, but still want to gather information from. In the case of transparency, you’d perform an alpha test on the texture of an object to see if you deliberately want to ignore the given object because the intersection point has an alpha value of zero.

As shown in Figure 2.6, this Intersection/Any Hit process continues until the closest hit is determined or no hit is found, with the *Closest Hit* shader and *Miss* shader being called respectively.

**Closest Hit**

The closest hit shader is performed on a piece of geometry that is determined to be the closest to the starting point of the ray, along the ray\(^2\). The shader code for a closest hit shader then is then similar to what one would write for a pixel shader (or fragment shader in OpenGL), including shading calculations like *Blinn-Phong* and returning the calculated value at the given point of intersection. Some complications arise here such as needing to perform manual interpolation for triangle geometry (since there is

\(^2\)However, it’s important to remember that using an *Any Hit* shader may complicate things by changing what is considered the ”closest” object.
no rasterization/interpolation step), but the core logic remains roughly the same for traditional lighting tasks.

Given the additional capabilities of ray tracing, it is also common to cast more rays (again, calling `TraceRay()`) from within this shader to calculate things like shadows, reflection, or refraction. Any additional ray "types" (for our contribution, we have primary and shadow ray types) require their own set of shaders for each piece of geometry. This allows something like a shadow ray to only check for intersection on hit, not run a full shading algorithm in its closest hit shader.

Miss

The Miss shader handles the case where a ray fails to intersect with any objects. Here, the ray could intersect a sky box to determine color value for an outdoors scene or set the background to a solid color.

2.3.3 Acceleration Structure

The acceleration structure can be considered the "engine" of the DXR pipeline. The ray generation stage of the pipeline feeds in rays for it to find objects along. We described what 3D acceleration structures typically look like above from an implementation level, however, DXR leaves the specifics of its acceleration structure to be vendor defined at the driver level[22]. Therefore, the only view into the acceleration structure is the small window provided via API itself.

Constructing the Acceleration Structure

Setting up the acceleration structure for DXR has similarities and dissimilarities to the rasterization equivalent of handling geometric information. As mentioned before, the acceleration structure must be built before dispatching rays to render a scene.
Therefore, it’s much more monolithic than a simple series of index and vertex buffers for a rasterization draw call.

The acceleration structure is effectively divided into two parts: the top-level and the bottom-level. The bottom-level acceleration structure holds individual geometric primitive information, such as vertex and index buffers for a triangle mesh. The top-level acceleration structure is the ”glue” between that geometric information and its associated shader table. This set-up also allows for instanced geometry using different shader tables, and therefore shading algorithms, using the same geometric information by having multiple top-level acceleration structures pointing to the same bottom-level acceleration structure [4].

**Using the Acceleration Structure**

The internals for the acceleration structure are obfuscated from the programmer to maximize any possible fixed-function optimization for the driver and hardware. So, interacting with the acceleration structure is an opaque process and boils down to passing it to the ray tracing pipeline without additional direct interaction within any shader. A notable exception performed on the CPU side is described below.

**Updating the Acceleration Structure**

In 3D application loops, two common steps are `Update()`, and `Render()`. The exact names and line between these two steps differ across applications. In our case the Render step is responsible for rendering one ray traced image from and presenting it to the screen. The Update step is responsible for updating all necessary information within our scene, including physics and animation. In a static scene, this step would then do nothing. However, 3D applications can have moving objects and this requires that the DXR acceleration structure have a mechanism to allow updates for it to be
practical.

Rigid body animation (animation via modifying the object’s transformation matrix) is achieved by modifying the transform supplied at the top-level acceleration structure level. This is relatively inexpensive and does not tax the acceleration structure much (see section 5 for our findings here). However, techniques such as skinned animation are more expensive than the rasterization equivalent due to needing the vertices to be transformed before being used in the raytracing pipeline (since all vertex locations must be known when you start tracing rays).\(^3\) This requires a separate vertex update step (usually in a compute shader [7]). We also need to update the acceleration structure somehow to acknowledge the change in vertex positions (remember the acceleration structure breaks down where objects are in 3D space; if the objects change their fundamental locations, it needs to be updated as well). As the acceleration structure is opaque, options here are limited.

We can either perform an incremental update or a full rebuild of the acceleration structure [7]. Updating an acceleration structure is much faster than performing a full rebuild, but comes with downsides. Notably, the 1.0 DXR functional specification states that an acceleration structure with the ability to update ”will not be as optimal in terms of raytracing performance” both before and after being updated [4]. Also, it ”can negatively impact raytracing performance, especially if the positions of the underlying objects have changed significantly from the original build of the acceleration structure before updates.”\(^4\) So, a DXR acceleration structure that uses an updating acceleration structures will be slower than those that don’t. The solution here is to minimize acceleration structure updates and perform a full rebuild when

\(^3\)The DXR specification actually calls out this exact issue of needing final vertex positions before build as a problem [4]. If this does end up being a common bottleneck for DXR applications, we would not be surprised if this functionality is optimized in future API versions.

\(^4\)While the acceleration structure internals are not specified, this could be due to the newly updated objects being taken out of the 3D acceleration structure and tracked separately. However, this is speculation.
performance allows (e.g. a scene or level change).
Chapter 3

RELATED WORKS

Raytracing is a well-researched area in computer graphics. Subtopics like global illumination are areas of constant improvement with researchers discovering faster and high quality ways to implement global illumination with raytracing. However, APIs like DirectX Raytracing have the benefit of tapping new hardware specifically designed for raytracing, leading to faster results while maintaining plausible quality. Because of that, this thesis focuses on enhancing the speed of global illumination. We therefore compare our contribution to similar contributions focused on fast, plausible approximations of global illumination. Below we present a few similar methods that have achieved similar results.

3.1 Monte Carlo Ray Tracing

We will first discuss the a straightforward raytracing global illumination technique: Monte Carlo raytracing. As per James Kajiya’s classic paper, *The Rendering Equation* [12], the contribution of indirect light can be calculated by taking the integral of all incoming light over the hemisphere of a surface (see Figure 3.1). Within the context of ray tracing one would need to fire an infinite number of rays to cover the entire hemisphere; this is not possible in practice. Thus, an approximation must be introduced.

Monte Carlo ray tracing does this approximation by simply limiting the number of rays used, typically between 32 and 512, for calculating indirect lighting contributions. These rays are then shot in random directions (within the dimensions of the surface’s hemisphere) which means there is a strong likelihood of a noisy result when fewer
Figure 3.1: A visual representation of Kajiya’s rendering equation. A portion of all incoming light over the hemisphere is reflected along the vector to the camera/eye. Monte Carlo global illumination is concerned with how we can approximate adding up all the incoming light at point \( x \), before we determine how much is reflected to the camera. CC BY-SA 3.0 [25].
samples are used in a complex scene (since two pixels next to one another will likely sample from different sets of rays). This noise can be reduced with various importance sampling and weighting techniques (e.g. cosine weighting, where rays closer to the normal are weighted to be more important in the sum calculation) but is an inherent problem of Monte Carlo ray tracing. Modern applications that use Monte Carlo ray tracing also typically use a denoising algorithm as a final pass to clean up and remove noise with minimal image quality loss.

3.2 Voxel Cone Tracing

Voxel cone tracing (VCT) is one of the best modern attempts to do real-time global illumination within the constraints of rasterization. However, it does borrow many ideas from raytracing (extending rays to ”cones”). Introduced in 2011 by Cyril Crassin, et al [3], VCT approximates indirect illumination by creating a voxelized representation of the scene geometry. Voxelization is the process of transforming a 3D object (typically a triangle mesh) into a 3D grid of voxels (named as a shortening of ”volumetric pixel”). This allows values to be easily stored in an octree or a 3D texture at a lower level of detail.

Following this idea, the voxelized representation is used to store approximated lighting values for each voxel of the scene. This structure is then mip-mapped to give layers of detail (important in the next step). We then want to calculate the final lighting values in the fragment shader like we normally do. To calculate the indirect contribution, multiple ”cones” are shot out in various directions and accumulate indirect lighting values from the voxel grid at set steps. This is assisted by using lower-detail mip-map levels as the distance along the cone increases (see image). This both keeps performance quick without degrading quality since indirect values further away should naturally contribute less with color bleeding (so a less accurate
Figure 3.2: An example render from the original voxel cone tracing application by Crassin, et al [3]. Note the full specular and diffuse reflections from the curtains in the Sponza scene.
value is okay). Further extensions include a separate, focused cone for specular values, and the possibility to calculate ambient occlusion values using this technique.

This technique, while relatively young, has seen use in commercial products [16]. However, it is an expensive technique even with the notable benefit of supporting dynamic objects. Still, voxel cone tracing is less computationally intensive than full ray traced global illumination and fits within the rasterization pipeline. The downsides to VCT stem from it being designed as an approximation. It is not designed to handle things like refraction and has issues with the voxel map being a rough approximation (especially at high mip levels). Sometimes crucial geometry can be skipped or missed when it is too small even if it should contribute indirect illumination (e.g. a mirror or thin wall).
Below we discuss the specifics regarding implementation of our novel ray traced global illumination approximation technique, focusing on the aspects pertinent to DXR. It is also important to note that we leveraged Microsoft’s DirectX Raytracing Sample repository for much of the initial ray tracing code [17]. While basic in support, it allowed us to iterate much more quickly on the novel parts of our contribution and gain a better understanding of how a DXR program should be organized. To run our application, a Windows 10 computer is required with the October 2018 Update, as well as hardware that supports the DXR API. These requirements could possibly change in the near future as DXR continues to mature as an API extension for DirectX.

4.1 Interacting with DirectX and DXR

DXR is designed to be an API extension for DirectX and can be used in conjunction with DirectX’s rasterization pipeline for hybrid rasterization/raytracing. Since our contribution is entirely around exploring DXR, we chose to only use raytracing in our rendering pipeline. This requires some setup that is unique from the normal rasterization DirectX 12 equivalent.

4.1.1 DirectX Raytracing pipeline

Before we can enter our rendering loop, we must perform some standard and non-standard DirectX setup. We start with the normal step of preparing our device and associated command list. The command list is necessary to submit commands to since DirectX 12 does not have an implicit context like previous versions of DirectX.
or an API like OpenGL.

After this initial setup we then start building our root signatures, where things begin to differ from standard DirectX 12. In DXR root signatures are put into two distinct groups: Global Root Signatures and Local Root Signatures. The global root signature’s arguments remain constant throughout the `DispatchRays()` call and include resources such as the output texture view (where the ray traced output will be stored), the acceleration structure, and any scene constant values (stored in a traditional constant buffer). The local root signature is then responsible for maintaining arguments unique to shader records within the shader table. These arguments will not only differ on a per-object basis, but will likely have different types of arguments entirely. For example, a triangle mesh’s local root signature contains the index and vertex buffer whereas a sphere local root signature contains its center position and radius. All of these arguments are then enumerated with unique shader register values (e.g. if triangle indices and vertices are stored in registers `t1` and `t2` respectively and used in triangle shaders, those shader registers cannot be reused to hold sphere related resources).

We then setup the raytracing pipeline state object which the documentation describes as ”representing a full set of shaders reachable by a `DispatchRays()` call, with all configuration options resolved, such as local signatures and other state”. In other words, this is how DXR knows which shader bytecode to use, and which ray types are associated with which shaders. This also links in the previously created global and local root signatures to their associated shaders and defines various metadata like maximum ray recursion depth (which theoretically allows the driver to potentially optimize around a smaller max depth). As a practical note for those interested in DXR, a lot of important linking and definition occurs here and its crucial to double-check things are setup correctly as even small bugs can be hard to track down later on (and can cause TDRs).
4.1.2 Uploading Geometric and Texture Resources

Next, geometry is loaded from file. Triangle mesh data is handled identically to DirectX 12 using index and vertex buffers. However, non-triangle geometry (spheres in our case) are all generically defined as AABB objects (axis-aligned bounding boxes). These work by defining the maximum space the object (sphere, box, or whatever geometry can be defined in a function) will occupy so the acceleration structure knows how to integrate it. Then the associated intersection shader does the final work to determine if a hit occurred or not (more on how we do that for a sphere below). We also upload this information to the GPU (for vertex and index buffers) and create the associated shader resource views (SRVs). SRVs effectively act like handles and provide metadata to DirectX when working with those resources (e.g. defining format for textures).

Now that our geometry has been loaded, we can proceed to build the acceleration structure. Our acceleration structure is comprised of individual bottom-level acceleration structure for each triangle mesh (e.g. the entire Sponza scene) and sphere (AABB defined geometry). We also use a single top-level acceleration structure for each bottom-level structure, except in cases where instancing is required for multiple of the same object.

4.1.3 Creating the Shader Table

The final setup portion of our application is creating the shader table that is indexed by DXR to apply the correct shaders and resources to each piece of geometry. This step links the shaders that were set up in the raytracing pipeline state object step with the rest of the shader record details. This includes local root signature arguments (textures, vertex buffers, index buffers, and more) as well as determining metadata like shader record size (which must be the maximum shader record size needed across
all shader records). As a linking step, most of the work done here is simple and
direct. However, like the pipeline state object creation step, bugs here are hard
to track down if introduced. Since the shader record table is indexed directly by
DXR once \textit{DispatchRays()} is called according to the offsets you give, bad offsets or
misconfigured shader tables can start failing in what appears to be random fashion
when DXR happens to index out-of-bounds or into a bad resource.

It’s important to have a debug mechanism to display the setup of the shader table
to verify its contents on the CPU (we found a simple print statement with an ASCII
table to suffice here). We also recommend investigating newer graphics debuggers
to verify shader table correctness and other DXR features. However, at the time of
development, graphics debuggers like Microsoft’s PIX were not compatible with DXR
when run on Nvidia’s Titan V due to a driver bug, so we were not able to use them
when developing our contribution.

4.2 Raytracing

Raytracing with DXR after the above setup is complete can be accomplished by
calling the function \textit{DispatchRays()}. As previously discussed in Section 2, this causes
DXR to shoot a ray for each point in the output location via a standard \textit{ray generation}
shader. It then tests for intersections within the acceleration structure. We primarily
rely on the built-in triangle intersection shader as the majority of our scene geometry
are triangle meshes. This is both fast and skips the process of needing to create
individual AABBs around triangle meshes like we do for spheres.

For intersecting spheres, we use a custom sphere intersection shader based around
the standard math equation for determining ray-sphere intersections. This is \textit{notice-
ably} slower than a triangle intersection shader, even with optimization (specifics are
discussed in the Section 5). Thus, we are judicious when deciding to use non-triangle
geometry.

As we do not use an *any hit* shader, the acceleration structure is free to iterate through objects in the scene using the current ray and returns the closest hit point via the *closest hit* shader or that a miss occurred via the *miss* shader. Assuming a hit did occur, we fall back to a simple *Phong* shading model for direct diffuse light in the scene for both triangle and sphere geometry. While more physically based, advanced lighting schemes exist, we chose to stick with *Phong* shading to keep the direct lighting scheme simple and fast, leaving most of the more advanced lighting work to the global illumination technique (described below). This also involves a texture access for both the diffuse color and normal map. This is cheap since this means only two texture look-ups per primary ray (more will likely be necessary with global illumination enabled).

Shadowing is done by casting another ray for every closest hit point from the hit position to each light. If no hit occurs, that light’s diffuse and specular color contributions are added. If there is a hit we can safely skip that light’s color contribution as the object is not visible from that light. The mechanism to trace this ray is identical to that found in the *ray generation* shader and only requires a *closest hit* and *miss* shader to return whether or not an object was hit with the given *shadow ray*.

If a no intersection occurs, a simple *miss* shader is invoked that returns a constant color as the background.

### 4.3 Monte Carlo Global Illumination with DXR

Performing global illumination is a good stress test of a ray tracing framework as it greatly increases the number of rays needed and requires many more intersections to occur. This tests DXR’s capability to produce rays and shade pixels accordingly, as well as the acceleration structure’s performance under stress. Our global illumination
technique tries to mesh the capabilities of DXR with standard Monte Carlo global illumination (see Section 3 for an introduction to Monte Carlo ray tracing). Most of this conversion was relatively straight-forward, with the majority of differences being structural. To that end, when a primary ray enters the closest hit shader we then shoot out a number of rays (between 2 and 256) using TraceRay(). These rays are the same as the primary ray and thus follow the same ray tracing pipeline discussed above (though the ray generation shader is not needed since that role is filled by the closest hit shader’s recursive TraceRay() call).

DXR then follows the exact same code that it did for the primary ray. The samples are randomly distributed along the hemisphere created via the surface’s normal. They are also cosine weighted according to their angle with the normal to increase the value of each sample (samples further from the normal’s direction are weighted less). The bounced light contribution is then accumulated and added to the direct diffuse light value.

4.4 Diffuse Spherical Approximation

The performance and quality impact of this global illumination step is discussed in greater detail in Section 5. In brief, it is prohibitively expensive to perform a full global illumination pass for diffuse light. Thus, our contribution attempts to reduce the cost of diffuse GI calculations by approximating geometry via a set of spheres. This idea was particularly designed for dynamic objects like hero characters (the player controlled character in a video game) whose lighting impact may be high on the look of a scene and can be approximated with spheres.

We approach this problem by using one or more spheres that overlay the dynamic triangle mesh (see Figure 4.1). This minimizes overlap of spheres while keeping the size of model similar. We then precompute a spherical texture of each segment
using a separate program that runs offline and apply the textures to their respective segmented spheres. The program we use was created by Bridget Winn and Christian Eckhardt and written in OpenGL [5]. These spheres are then integrated into the DXR pipeline via custom AABB-defined geometry and a sphere intersection shader that follows the traditional ray-sphere intersection equation. While a single sphere is obviously less complex than the potentially thousands of triangles, an important trade-off is that the custom sphere intersection shader is going to be necessarily slower than the fixed-function triangle intersection shader provided by DXR and backed up by the GPU vendor drivers. However, assuming that the triangle geometry segments are sufficiently complex and can be segmented into spheres without waste (e.g. a wall would be a poor choice due to low triangle density and low object volume), we would expect some sort of performance win. This is discussed further in Section 5.

While we want to use the approximate geometry for the bounced GI light, we still want to use the high detail character mesh for our primary rays. This requires using two separate acceleration structures, one that contains the full triangle mesh and one that replaces the mesh with the spherical approximation. This is accomplished by

Figure 4.1: The left picture shows a normal boot triangle mesh. The left picture is that same boot converted to a spherical texture representation and applied to a sphere. The sphere is necessarily larger than the boot in areas due to the boot not being perfectly round.
building two acceleration structures at setup time which both exist on the GPU. These acceleration structures share the majority of their data to avoid wasting CPU/GPU cycles and memory (i.e. both share the same bottom-level acceleration structures for the static Sponza geometry). The only difference is that the GI acceleration structure has the sphere AABB top-level acceleration structure versus the full triangle character mesh.

At the shader level, we then pass the standard full-scene acceleration structure when tracing the primary and shadow rays to \textit{TraceRay()} and the modified GI acceleration structure when calling \textit{TraceRay()} for the bounced GI rays. This successfully applies the approximate sphere geometry to the bounced light calculation, bypassing the complicated triangle mesh for bounced ray.

4.5 Note on Denoising

We decided against implementing a denoising technique for improving the image quality post Monte Carlo global illumination. While denoising is critical to minimizing the number of samples required for good image quality when using Monte Carlo raytracing we didn’t want to focus our efforts on a topic that is both rapidly advancing and requires significant investment to do right. However, our contribution would work with denoising techniques (typically run in image-space) and then use fewer samples to achieve equivalent or better image quality.
Chapter 5

RESULTS

Below we present an overview as well as a detailed discussion of the results of this thesis.

5.1 Experimental Setup

For our tests we used a modified version of the classic Crytek Sponza model (available via [14]) as the base model for the scene. We also included a few additional models used for non-static geometry [26, 8]. While few in number, this provided a reasonably complex scene to test our contributions without needing additional support for custom assets or more complicated asset loading schemes. Specific polygon counts are provided in Table 5.1.

We also relied on a Nvidia Titan V graphics card (provided by California Polytechnic State University and the Mixed Reality Lab [5]) which was the first graphics card to support DXR. Other cards with DXR support such as the RTX 2080 are untested due to not having access to these cards (though we expect similar results on

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Number of Polygons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Sponza</td>
<td>262,267</td>
</tr>
<tr>
<td>Old Man</td>
<td>10,518</td>
</tr>
<tr>
<td>Old Man (Higher Resolution)</td>
<td>723,168</td>
</tr>
<tr>
<td>Boot</td>
<td>634</td>
</tr>
</tbody>
</table>

Table 5.1: The polygon counts for each triangle mesh used in the project. Spherical geometry is not included as they are not triangle meshes.
these cards inline with their performance profiles. All tests were run on Windows 10 with the October 2018 Update (version 1809) due to a dependency on DXR API version 1.0 which was released in that update [28].

5.2 Analysis

Below we briefly break down the performance of DXR and its relative performance impact. We then look at the various performance and quality profiles of our GI implementation and spherical approximation improvement.

5.2.1 DXR Performance

In Table 5.2, we have a series of timings with only the static Sponza scene (no dynamic meshes) to provide a baseline performance. We give timings based on profiles varying resolution and GI sample counts. Overall, we can see that enabling GI incurs a heavy cost, even without dynamic geometry. Having even 8 GI rays for each primary ray cuts performance by a factor of ten at the lowest resolution alone.

To investigate this further we added more dynamic ”Old Man” character mesh instances (multiple top-level acceleration structure) to see how performance would be impacted. These results are in Table 5.3 with an example render is in Figure 5.3. The old man model was approximated with 15 spheres. It’s interesting to compare these results to the Sponza only performance. While the higher model counts (100+) see significant performance decreases, we see that the 10 mesh results are actually

---

1An important note is that we leave in a GPU synchronization call to simplify the Acceleration Structure update logic (which is only necessary with dynamic objects). This ensures consistency between our timings for dynamic and non-dynamic objects as this call could be removed entirely in future work. Without the synchronization, any non-GPU bottlenecked framerates (specifically the ”No GI” column) can increase dramatically, up to around 800fps at the lowest resolution. However this artificial bottleneck does not affect the GI numbers due to needing to wait on the GPU regardless as it becomes the bottleneck naturally. Being that those timings are the focus, we find this solution acceptable.
Table 5.2: Frame rates (frames per second) when only the Sponza scene is included averaged over 5 seconds (all FPS counts are averaged over 5 seconds, with FPS polled once a second). Times marked N/A failed to run due to a GPU hang (TDR, or related crash). Rendered frame is shown in Figure 5.1.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>No GI</th>
<th>8 Samples</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>640 x 480</td>
<td>193.43</td>
<td>18.04</td>
<td>9.31</td>
<td>4.75</td>
<td>2.40</td>
<td>1.21</td>
<td>&lt; 1.0</td>
</tr>
<tr>
<td>1280 x 720</td>
<td>190.60</td>
<td>6.51</td>
<td>3.29</td>
<td>1.65</td>
<td>&lt; 1.0</td>
<td>N/A</td>
<td>-</td>
</tr>
<tr>
<td>1920 x 1080</td>
<td>101.81</td>
<td>3.05</td>
<td>1.51</td>
<td>&lt; 1.0</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 5.1: A 1920 x 1080 resolution render of the Sponza scene using our application. No dynamic geometry is included but GI is enabled with 32 samples and a single GI bounce. This shot of our scene is what we used for all our benchmarks unless otherwise noted.
higher than the base Sponza results (as well as the single mesh results in some cases). Similar to how the number of fragments being rendered in a rasterization pipeline can affect performance heavily, the size and complexity of geometry visible in the scene can affect raytracing performance. Exactly how is hard to say without deeper knowledge of the acceleration structure.

<table>
<thead>
<tr>
<th>Number of Meshes</th>
<th>No GI</th>
<th>8 Samples</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>194.68</td>
<td>18.51</td>
<td>9.39</td>
<td>4.87</td>
<td>2.46</td>
<td>1.22</td>
<td>&lt; 1.0</td>
</tr>
<tr>
<td>10</td>
<td>190.38</td>
<td>25.80</td>
<td>13.83</td>
<td>7.04</td>
<td>3.62</td>
<td>1.78</td>
<td>&lt; 1.0</td>
</tr>
<tr>
<td>100</td>
<td>134.81</td>
<td>13.63</td>
<td>7.14</td>
<td>3.64</td>
<td>1.85</td>
<td>&lt; 1.0</td>
<td>N/A</td>
</tr>
<tr>
<td>1000</td>
<td>25.98</td>
<td>1.94</td>
<td>&lt; 1.0</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>3.27</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.3: This table measures FPS with the Sponza scene and various amounts of the ”Old Man” character mesh. Resolution is held constant at 640 x 480. The sphere approximation method is not used for the GI measurements.

Figure 5.2: Graph of the data from Table 5.3.
Figure 5.3: An example image of adding the "Old Men" as dynamic objects to the scene. There are 10 men evenly spaced here. When the number of men is increased, the length covered remains the same but the density of objects increases.

5.2.2 Global Illumination Performance

With some benchmarks for how DXR performs, we will now discuss the impact of adding in full diffuse global illumination. As previously mentioned, our GI uses a user chosen 8-256 rays with randomly chosen directions over the normal hemisphere of the hit surface. Only a single bounce is used. Increasing the number of bounced rays per primary ray results in increased image quality with a less noisy image. As Table 5.2 shows, performance degrades rapidly as the number of GI rays are increased even as the resolution is held constant. This makes sense as each additional GI ray effectively adds another whole screen resolution number of rays as each primary ray must trace an additional ray. Image quality differences are shown in Figure 5.4. As mentioned
in Section 4, an image denoiser would reduce the number of rays required for a higher image quality and is left for future work.

Figure 5.4: The effect of enabling diffuse global illumination is immediately apparent between the images with and without it. Increasing the sample count from 8 to 32 begins to clean up the image noise (though it is still noticeable).
5.2.3 Impact of Spherical Diffuse Global Illumination Approximation

Now that we have a good idea on the performance of DXR with our scene from a performance and aesthetic perspective, we can take a look at the impact of the spherical approximation technique.

Performance

In Table 5.4, we include FPS measurements when using our spherical approximation method with varying numbers of dynamic objects. From this we can see that performance is mixed. When the mesh count is 1 and 10, the FPS of the application does increase from anywhere between 1% to 12%. However, as the mesh count increases to 100 and 1000, we see an FPS *loss* that roughly matches the percentage gained in the previous lower mesh count tests. This result was perplexing and required further investigation.

<table>
<thead>
<tr>
<th>Number of Meshes</th>
<th>No GI</th>
<th>8 Samples</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>194.68</td>
<td>20.08</td>
<td>10.44</td>
<td>5.36</td>
<td>2.71</td>
<td>1.35</td>
<td>&lt; 1.0</td>
</tr>
<tr>
<td>10</td>
<td>190.38</td>
<td>27.17</td>
<td>14.52</td>
<td>7.26</td>
<td>3.73</td>
<td>1.87</td>
<td>&lt; 1.0</td>
</tr>
<tr>
<td>100</td>
<td>134.81</td>
<td>12.37</td>
<td>5.99</td>
<td>3.05</td>
<td>1.54</td>
<td>&lt; 1.0</td>
<td>N/A</td>
</tr>
<tr>
<td>1000</td>
<td>25.98</td>
<td>1.85</td>
<td>&lt; 1.0</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>3.27</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.4: This table measures FPS with conditions equivalent to Table 5.3. However, it uses the sphere approximation method for GI measurements as an exception.

Our first idea to figure out the problem was to increase the polygon count of the "Old Man" model to a much higher degree (as seen in Table 5.1). This higher level of detail model would theoretically benefit even more from the spherical representation since it would require a more complicated set of intersection tests as the triangle
Figure 5.5: Graph of the data from Table 5.4.

Figure 5.6: Graph of the change in FPS values from Table 5.3 to 5.4. All comparison charts leave off data points where zero change occurred (i.e. when no GI samples are used), or when one or both tables have incomplete data (fields marked ‘N/A’ or ‘-’).

density is higher for the same space compared to the lower detail model. While FPS dropped to varying degrees across the board, the results were still inline with what occurred in Table 5.4 (e.g. for 10 meshes, no GI went from 190.38fps to 92.68. The difference in performance for the GI tests was less, with 8 samples only being 1fps slower). However, the low drop in FPS for GI tests indicated that intersection tests
were not the bottleneck for our application. To better understand the real impact of the spherical approximation, we needed to remove the current bottleneck as much as possible.

To that end, the amount of work performed during different portions of a DispatchRays() call is varied and hard to measure since so many shader types and shader implementations can run in any given call. So, determining the bottleneck was difficult outside of removing bits of code and rerunning the application manually. To try and reduce the ”noise” around our spherical approximation (and hopefully the bottleneck), we opted to remove as much logic as possible from the closest hit shaders for both spheres and triangles that would only be run in GI situations (e.g. instead of creating a randomly directed secondary GI ray within the normal’s hemisphere, we just set the ray to point along the (1, 1, 1) vector). This removes the GI functionality that improves scene quality, but allows us to compare the impact of spherical approximation intersections versus the normal triangle intersections.

We then ran the same dynamic object tests used for Tables 5.3 and 5.4 with our newly streamlined shader code. These results are in Tables 5.5 and 5.6. Here we begin to see that using our sphere approximation has the potential to be faster than the raw triangle mesh when enough meshes are used. The 10 mesh test case sees performance increases between 7.3% and 13.3% as an example. These numbers are much more inline with our original expectations for using spheres as approximations for geometry during secondary rays. So, like any optimization, for our contribution to have a noticeable effect, the intersection performance must be the bottleneck in the application. In our case, the bottleneck ended up being the closest hit shaders (triangle and sphere) that performed the actual shading calculations, so we didn’t see noticeable improvement.

We then revisited our higher resolution ”Old Man” model to see if using it could
Table 5.5: This table measures FPS with conditions equivalent to Table 5.3. However, we’ve removed most of the Closest Hit shader logic from secondary bounces to try to narrow performance differences to the intersection logic.

<table>
<thead>
<tr>
<th>Number of Meshes</th>
<th>No GI</th>
<th>8 Samples</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>194.68</td>
<td>108.43</td>
<td>70.56</td>
<td>41.88</td>
<td>24.49</td>
<td>13.27</td>
<td>6.88</td>
</tr>
<tr>
<td>10</td>
<td>190.38</td>
<td>74.97</td>
<td>46.53</td>
<td>28.03</td>
<td>15.57</td>
<td>8.20</td>
<td>4.22</td>
</tr>
<tr>
<td>100</td>
<td>134.81</td>
<td>26.30</td>
<td>15.77</td>
<td>8.46</td>
<td>4.33</td>
<td>2.24</td>
<td>1.12</td>
</tr>
<tr>
<td>1000</td>
<td>25.98</td>
<td>3.67</td>
<td>1.97</td>
<td>1.04</td>
<td>&lt; 1.0</td>
<td>N/A</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>3.27</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 5.7: Graph of the data from Table 5.5.

positively impact our results. Having more triangles per sphere would hopefully increase the performance gain of using spheres instead. These results are in Tables 5.7 and 5.8. Using the 10 mesh test case as an example again, we see performance increases between 13.5% and 39.0%. This is an even larger improvement than the lower polygon model and is exactly the kind of improvement that was expected. Seeing the performance gain increase as the number of GI samples increased also validates our theory (since more and more of the rays traced depend on the spherical intersection).
<p>|</p>
<table>
<thead>
<tr>
<th>Number of Meshes</th>
<th>No GI</th>
<th>8 Samples</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>194.68</td>
<td>105.65</td>
<td>71.33</td>
<td>42.85</td>
<td>24.84</td>
<td>13.36</td>
<td>7.15</td>
</tr>
<tr>
<td>10</td>
<td>190.38</td>
<td>82.92</td>
<td>49.91</td>
<td>30.28</td>
<td>17.19</td>
<td>9.19</td>
<td>4.78</td>
</tr>
<tr>
<td>100</td>
<td>134.81</td>
<td>26.79</td>
<td>15.96</td>
<td>8.63</td>
<td>4.48</td>
<td>2.33</td>
<td>1.18</td>
</tr>
<tr>
<td>1000</td>
<td>25.98</td>
<td>3.89</td>
<td>2.11</td>
<td>1.12</td>
<td>&lt; 1.0</td>
<td>N/A</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>3.27</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.6: This table measures FPS with conditions equivalent to Table 5.4. However, we’ve removed most of the Closest Hit shader logic from secondary bounces to try to narrow performance differences to the intersection logic.

Figure 5.8: Graph of the data from Table 5.6.

So, as an application begins to add more and more triangle density to their scene, using our contribution is more likely to improve performance if intersections are the bottleneck in their application.

After running both sets of tests for our high and low polygon models, it’s clear that our method of approximating geometry with sphere’s for secondary GI ray intersections does provide significant improvement. However, our application saw mixed results due to a bottleneck within our closest hit shaders.
Figure 5.9: Graph of the change in FPS values from Table 5.5 to 5.6.

<table>
<thead>
<tr>
<th>Number of Meshes</th>
<th>No GI</th>
<th>8 Samples</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>84.35</td>
<td>73.17</td>
<td>54.43</td>
<td>35.91</td>
<td>21.72</td>
<td>12.11</td>
<td>6.78</td>
</tr>
<tr>
<td>10</td>
<td>77.25</td>
<td>55.84</td>
<td>35.28</td>
<td>21.44</td>
<td>12.10</td>
<td>6.44</td>
<td>3.28</td>
</tr>
<tr>
<td>100</td>
<td>74.91</td>
<td>21.37</td>
<td>12.99</td>
<td>6.94</td>
<td>3.72</td>
<td>1.91</td>
<td>&lt; 1.0</td>
</tr>
<tr>
<td>1000</td>
<td>23.48</td>
<td>3.42</td>
<td>1.76</td>
<td>&lt; 1.0</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>3.27</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.7: This table measures FPS with conditions equivalent to Table 5.5. However, it uses the higher resolution old man model described in Table 5.1.

Quality

Looking at the quality of the spherical approximation we take an up-close look at our boot model in a few different orientations. Figures 5.13 and 5.14 both show direct pixel for pixel image comparisons. Overall, we see differences between using the spherical approximation and the full boot mesh for bounced diffuse light. The exact differences and their impact discussed further the caption for each figure.
Table 5.8: This table measures FPS with conditions equivalent to Table 5.6. However, it uses the higher resolution old man model described in Table 5.1.

To summarize those results, the spherical approximation appears to be a worse choice for more concave geometry as it struggles with the non-convex aspects of the boot. The pre-computed spherical texture is also an imperfect representation of the top of the boot causing differences along due to too much orange bounced light.
Figure 5.11: Graph of the data from Table 5.8.

Figure 5.12: Graph of the change in FPS values from Table 5.7 to 5.8.
Figure 5.13: Here we compare the result of using spherical approximation of the boot with the full mesh model for GI. Notably it appears that the orange accent along the top of the boot is over-represented in the spherical model causing too much bounced orange light at the top of the pillar (the upper pink area). On the other hand, the concave toe of the boot fails to bounce significant orange light on the lower portion of the column (the lower pink area). The downside of only using one bounce of global illumination is that some areas remain completely dark.
Figure 5.14: Following from the previous comparison in Figure 5.13, we again see the orange trim being over-represented along the top of the boot. Overall, this angle gets better results than the previous comparison as the back of the boot is not concave, which caused issues in the previous image. We also see minor differences along the boot due since it won’t receive ”self-inflicted” bounced light contributions in the approximation like in the full mesh.
Figure 5.15: More comparisons following Figures 5.13 and 5.14. Here the boot is placed in the corner of two white walls with the light shining perpendicular to the boot. Noise is much more pronounced due to the simple background, but it is easier to see the differences described in the previous figures.
Figure 5.16: A similar shot to Figure 5.15 but all lighting has been removed except for diffuse GI from the boot to maximize visibility.
In this thesis we presented a novel approach to approximating diffuse global illumination using the new DirectX Raytracing API. We feel that it is a good starting point for further experimentation and improvements, even if the performance and quality of our results were mixed. The effectiveness of the solution varies with the complexity and shape of the mesh being approximated. It also heavily depends on whether geometry intersection is the bottleneck in the application. We also feel that with hardware or driver support for a sphere intersection shader, the results could be improved even further as the built-in triangle intersection shader is faster than what can be obtained with a custom intersection shader [4]. We look forward to future research that takes advantage of unique DXR features, like the intersection shader, in new and inventive ways.

6.1 Future Work

While we are happy with the work completed and the results obtained with this thesis, there is notable room for improvement. One area is the general structure and performance of the our DXR related code. This thesis was the first time we had worked with DXR and in future work many things would be done differently. For example, having better abstractions for bottom and top-level acceleration structures would have allowed more flexibility when using different acceleration structures for primary and bounced rays. Rethinking some of the core code would make it easier to both extend and learn from in the future. We also used an individual AABB for each sphere. The DXR specification cautions against "bounding volume bloating"
(where you stuff too much geometry in a single AABB) as performance impacting [4]. However, smarter packing of AABBs with multiple spheres or other geometry types may be worth investigating.

Moving on, performing further work on the GI solution could be beneficial. This includes things like specular reflections, caustics, and refraction to improve overall image quality and provide a more realistic baseline for further research. Diffuse global illumination, although important to improving the realism of a scene, is often subtle. Adding in additional support for other global illumination features would greatly add to the realism of the scene and better showcase our contribution in a more production-like environment. A further extension of this idea would be to implement a full path tracer with DXR including our contribution. While likely slower, this would provide a better ground-truth to compare our current solution against. On the opposite end of the spectrum, implementing some kind of hybrid rasterization/raytracing renderer would allow for interesting performance and quality comparisons. Specifically, doing the direct lighting pass via rasterization and the diffuse GI as part of a second DXR pass.

Finally, further improvements to the spherical texture approach, including optimizations in DXR shader code and how the spherical texture is created would further improve the quality and performance of the spherical approximation without hardware changes. Also, continuing to optimize our closest hit shaders to make it less of a bottleneck for performance. To that end we also struggled to find open source DXR Monte Carlo Raytracing programs similar to ours. Having comparable benchmarks would be valuable to validate our findings against.
BIBLIOGRAPHY


[8] G. Harris. Workman steelcap rubber boot. Licensed under the TurboSquid Royalty Free License by
https://blog.turbosquid.com/royalty-free-license/.


[26] P. Tosca. Old man. Licensed under the TurboSquid Royalty Free License by
[27] M. Valient. Taking killzone shadow fall image quality into the next generation. 


[29] WhiteTimberwolf. Octree2. Licensed under CC BY-SA 3.0 by 
https://creativecommons.org/licenses/by-sa/3.0/deed.en with no changes. 

Licensed under CC BY-NC-ND 4.0 by 
https://creativecommons.org/licenses/by-nc-nd/4.0/ with no changes. 
Accessed via 