AN ANALYSIS OF REAL-TIME RAY TRACING TECHNIQUES USING THE VULKAN® EXPLICIT API

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
Elleis C. Souza
June 2021
COMMITTEE MEMBERSHIP

TITLE: An Analysis of Real-Time Ray Tracing Techniques Using the Vulkan® Explicit API

AUTHOR: Elleis C. Souza

DATE SUBMITTED: June 2021

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

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

COMMITTEE MEMBER: Chris Lupo, Ph.D.
Professor of Computer Science
ABSTRACT

An Analysis of Real-Time Ray Tracing Techniques Using the Vulkan® Explicit API
Elleis C. Souza

In computer graphics applications, the choice and implementation of a rendering technique is crucial when targeting real-time performance. Traditionally, rasterization-based approaches have dominated the real-time sector. Other algorithms were simply too slow to compete on consumer graphics hardware. With the addition of hardware support for ray-intersection calculations on modern GPUs, hybrid ray tracing/rasterization and purely ray tracing approaches have become possible in real-time as well. Industry real-time graphics applications, namely games, have been exploring these different rendering techniques with great levels of success. The addition of ray tracing into the graphics developer’s toolkit has without a doubt increased what level of graphical fidelity is achievable in real-time.

In this thesis, three rendering techniques are implemented in a custom rendering engine built on the Vulkan® Explicit API. Each technique represents a different family of modern real-time rendering algorithms. A largely rasterization-based method, a hybrid ray tracing/rasterization method, and a method solely using ray tracing. Both the hybrid and ray tracing exclusive approach rely on the ReSTIR algorithm for lighting calculations. Analysis of the performance and render quality of these approaches reveals the trade-offs incurred by each approach, alongside the performance viability of each in a real-time setting.
ACKNOWLEDGMENTS

Thanks to:

• My advisor, Dr. Zoë Wood, for introducing me to computer graphics.

• The members of the graphics group, for their wealth of knowledge and willingness to learn new things.

• My parents, Mark and Dianne, and sister, Avery. Completing this thesis would have been impossible without your unwavering love and support.

• My friends, for never failing to provide distractions.
# TABLE OF CONTENTS

| LIST OF TABLES | ix |
| LIST OF FIGURES | x |

## CHAPTER

1. **Introduction**
   - 1.1 Rendering Engine
   - 1.2 Outline

2. **Background**
   - 2.1 Rendering Algorithms
     - 2.1.1 Rasterization
     - 2.1.2 Ray Tracing
   - 2.2 Vulkan
   - 2.3 Real-Time Ray Tracing
   - 2.4 Physically Based Rendering
     - 2.4.1 Materials
     - 2.4.2 Direct & Indirect Illumination

3. **Related Works**
   - 3.1 RTX Algorithms
   - 3.2 ReSTIR
     - 3.2.1 Reservoirs
     - 3.2.2 Temporal Reuse
     - 3.2.3 Spatial Reuse
     - 3.2.4 The Algorithm
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Pros and cons of the rasterization algorithm</td>
</tr>
<tr>
<td>2.2</td>
<td>Pros and cons of the ray tracing algorithm</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A flowchart depicting the graphics pipeline, adapted from [25].</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>A non-converged (left) and converged (right) image of the Sponza scene, rendered using the Blender Cycles renderer.</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>The RTX programming model [9].</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>A render before and after a denoising step [24].</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Depiction of the microfacet model [33].</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>The rendering equation [19]. Descriptions of each variable can be found in the appendix.</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>The rendering equation for direct lighting, summing over the unit hemisphere. Descriptions of each variable can be found in the appendix.</td>
<td>17</td>
</tr>
<tr>
<td>2.8</td>
<td>The rendering equation for direct lighting, summing over light emitters. Descriptions of each variable can be found in the appendix.</td>
<td>18</td>
</tr>
<tr>
<td>2.9</td>
<td>One render including only direct lighting, and another including both direct and indirect [29].</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>The various components that contribute to a render in SEED’s PICA PICA [14].</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>A screenshot from Minecraft RTX [13].</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>An image of the zeroday scene, rendered using the ReSTIR implementation from [16].</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>An example of temporal backprojection.</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>An example of temporal and spatial reuse in the Subway scene, from [16].</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>An example of metallic-roughness textures, from [21].</td>
<td>31</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------</td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>A summary of the metallic-roughness BRDF [2].</td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>A chart depicting the influence of the Disney BRDF parameters [17].</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>An overview of the renderer’s architecture. Blue tiles are explained further in subsequent figures.</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>A detailed look at the BufferManager singleton.</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>A detailed look at the AssetManager singleton and the associated Model class.</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>A detailed look at the DescriptorManager singleton.</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>A detailed look at the CommandManager singleton.</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>A detailed look at the QueueManager singleton.</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>A detailed look at the PresentationManager singleton.</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td>A detailed look at the abstractions provided by the Renderer class.</td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td>A render depicting the effect of changing metallic and roughness values.</td>
<td></td>
</tr>
<tr>
<td>4.10</td>
<td>A render of the “damaged helmet” glTF model utilizing all five material maps.</td>
<td></td>
</tr>
<tr>
<td>4.11</td>
<td>An overview of the metallic-roughness physically-based rendering style.</td>
<td></td>
</tr>
<tr>
<td>4.12</td>
<td>A convoluted (left) and filtered (right) side of the environment map.</td>
<td></td>
</tr>
<tr>
<td>4.13</td>
<td>A render of the Sponza scene with the full PBR rasterization rendering style.</td>
<td></td>
</tr>
<tr>
<td>4.14</td>
<td>An overview of the Hybrid ReSTIR rendering style.</td>
<td></td>
</tr>
<tr>
<td>4.15</td>
<td>Renders of some G-Buffer targets for the Sponza scene.</td>
<td></td>
</tr>
<tr>
<td>4.16</td>
<td>A render of the Sponza scene without (left) and with (right) temporal reuse.</td>
<td></td>
</tr>
</tbody>
</table>
4.17 A render of the Sponza scene without (left) and with (right) spatial reuse. The effect is particularly notable around harsh shadow edges. ........................................... 55

4.18 An overview of the RT-Only ReSTIR rendering style. ............. 56

4.19 A render of the Sponza scene with the RT-Only ReSTIR 180° Fisheye Lens, without second bounce lighting. ....................... 58

4.20 A render of the Sponza scene with the RT-Only ReSTIR 180° Fisheye Lens, including second bounce lighting with a gaussian blur. ......................................................... 59

5.1 A metallic-roughness rasterization render of a suit of armor with a studio environment map, shadows disabled. ....................... 61

5.2 A metallic-roughness rasterization render of a damaged helmet with a park environment map, shadows disabled. ..................... 61

5.3 A metallic-roughness rasterization render of the Sponza scene with a park environment map, shadows enabled. ....................... 62

5.4 A metallic-roughness rasterization render of the San Miguel scene with a park environment map, shadows enabled. ..................... 62

5.5 A Hybrid ReSTIR render of the Sponza scene with 15360 triangle emitters.................................................. 63

5.6 A Hybrid ReSTIR render of the Sponza scene with a lamp containing 15360 triangle emitters. ................................. 64

5.7 A Hybrid ReSTIR render of the Sponza scene with 46080 triangle emitters, divided equally between 3 different colored spheres. ... 64

5.8 A Hybrid ReSTIR render of the San Miguel scene lit by an orange torus containing 18432 emissive triangles. ....................... 65

5.9 A 180° RT-Only ReSTIR render of the Sponza scene with 15360 triangle emitters and single-bounce indirect lighting. ............... 66

5.10 A 180° RT-Only ReSTIR render of the Sponza scene with a lamp containing 15360 triangle emitters, including single-bounce indirect lighting. .................................................. 66
5.11  A 180° RT-Only ReSTIR render of the Sponza scene with 46080 triangle emitters (divided equally between 3 different colored spheres), including single-bounce indirect lighting. ................................. 67

5.12  A 180° RT-Only ReSTIR render of the San Miguel scene lit by an orange torus containing 18432 emissive triangles, including single-bounce indirect lighting. ........................................... 67

5.13  A plot depicting the number of rasterized triangles vs. frametime using the PBR rasterization method. ................................. 69

5.14  A plot depicting the number of triangles in the scene vs. frametime using the Hybrid ReSTIR method. ................................. 69

5.15  A plot depicting the number of triangles in the scene vs. frametime using the RT-Only ReSTIR method. ................................. 70

A.1  The rendering equation [19] with variable definitions. Note that the same definitions apply for the direct lighting and area rendering equations as well. ................................. 78
Chapter 1

INTRODUCTION

Real-time computer graphics applications have largely made use of the same set of tools since the early 2010s. In the last few years, however, advancements in graphics hardware and software have led to a paradigm shift of sorts — algorithms that were once unfeasible to perform in real-time have rapidly increased in speed. *Minecraft RTX, Cyberpunk 2077, and Call of Duty: Black Ops Cold War*, games with exceptional graphics quality, are all built on these recent technological advancements.

The approaches taken by developers that make use of these new technologies are varied and nuanced. Achieving desired visual effects is a complex and application-specific process; the blending of industry-tested techniques and new tools being crucial for staying on the cutting edge.

1.1 Rendering Engine

In this thesis, a rendering engine was developed to support both classical and cutting-edge image-synthesis techniques. The express goal of this engine was to allow for the development of and comparison between families of rendering approaches being used in different segments of real-time graphics applications.

Three rendering approaches are supported, each representing a distinct approach to make use of modern day graphics hardware/software.

The families represented by these approaches can be classified as (1) purely classical, (2) hybrid classical/experimental, and (3) purely experimental. The terms “classical”
and “experimental” each describe distinct rendering algorithms, namely rasterization and ray tracing respectively.

Real-time rendering engines are generally built on a library of supporting graphics software. Very few of these libraries have support for cutting edge advancements, with only DirectX 12 and the Vulkan® Explicit API being potential candidates for this engine. The Vulkan® Explicit API was selected, resulting in a cross-platform engine capable of supporting all three families of rendering methods analyzed in this thesis.

For all three rendering approaches, real-time frame rates were achieved with over 10 million triangles present in the scene. Development and testing of this engine occurred on a Windows 10 desktop with a NVIDIA GeForce RTX 2070 Super GPU and AMD Ryzen 3800x CPU.

1.2 Outline

Chapter 2 provides an overview of the algorithms and history of relevant rendering techniques. In addition, an overview of lighting calculations, materializing, and graphics APIs is provided. Next, Chapter 3 details specific techniques and mathematical models leveraged within this thesis’ rendering engine. The actual implementation of the engine is detailed in Chapter 4, with the development decisions for the engine itself alongside the three rendering techniques explained here. Renders and performance metrics from the rendering techniques are contained in Chapter 5, and possible future improvements to the engine are proposed in Chapter 6.
Chapter 2

BACKGROUND

2.1 Rendering Algorithms

The process of synthesizing an image from a set of input geometry is known as rendering. When rendering, regardless of the specific algorithm used, the goal is to determine the color of each pixel. Many different families of rendering algorithms exist, with two of the most popular being rasterization and ray tracing. Although historically these algorithms have been used separately, recently there has been a push towards blending these techniques in contexts such as real-time rendering.

An explanation of those two techniques will be outlined in the following sections, with the caveat that only the case of rendering triangles will be considered.

2.1.1 Rasterization

In the context of real-time rendering, the de-facto rendering approach has always been rasterization. Rasterization is known as an object-centric algorithm, as it starts
by selecting a triangle, and then figures out which pixels in the image it occupies (if any). The pseudocode for the algorithm is provided in Algorithm 1.

Algorithm 1: Pseudocode for the rasterization algorithm

Result: A colored array of pixels

for every triangle in the scene do
   Project the triangle into image space;
   for every pixel in the output image do
      if the pixel is in the projected triangle then
         color the pixel;
      end
   end
end

Because drawing each triangle into the image is an independent process, this algorithm lends itself extremely well to a pipelined, parallelized environment. The “graphics pipeline” that GPUs support is a rasterization-based rendering algorithm. This pipeline has hardware support for each of the steps in the rasterization algorithm, such as constructing triangles from vertices, projecting the triangles into image space, and clipping triangles that are outside of the image being rendered. Unfortunately, one downside of the low-computation cost of rasterization is that it does not easily support transmissive materials, reflections, or realistic shadowing (all of which are desirable in photorealistic renders). A detailed look at the modern day rasterization-based graphics pipeline is presented in Figure 2.1.

Compared with other techniques available when the first dedicated GPUs were being developed, rasterization was the clear choice for what algorithm to include hardware support for. It is easy to translate into a parallelized environment and has a relatively low computational cost per triangle. The popularity of real-time computer graphics
Figure 2.1: A flowchart depicting the graphics pipeline, adapted from [25].

Applications (which heavily utilize the GPU) have spurred a substantial amount of research into techniques for achieving certain visual effects that are difficult to do in a rasterization setting. Shadow penumbra, dielectrics, reflections, and more are all possible to do in a rasterization pipeline now; however, the computational cost for these techniques adds up quickly. Often, this is due to a lack of built-in methods for performing spatial queries in rasterization-based frameworks.

Table 2.1: Pros and cons of the rasterization algorithm

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quite fast</td>
<td>Transparency is difficult to achieve</td>
</tr>
<tr>
<td>Easy to parallelize and pipeline</td>
<td>Computational cost of shadows and shadow penumbra is high</td>
</tr>
</tbody>
</table>
2.1.2 Ray Tracing

Applications striving for photorealism will often use the Ray Tracing algorithm. This algorithm has the ability to generate extremely photorealistic and detailed images natively without the addition of complex techniques. Ray tracing is an image-centric rendering algorithm, in which you start at an image pixel, and then shoot rays out to determine what geometry that pixel will display. The pseudocode for the algorithm is provided in Algorithm 2.

Algorithm 2: Pseudocode for the rasterization algorithm

Result: A colored array of pixels

for every pixel in the image do
    Cast a ray out into the scene;
    for every triangle in the scene do
        Determine the closest triangle the ray hits;
    end
    Shoot another ray from the intersection point, or return the calculated color of the intersected material;
end

The geometry of the scene is usually stored in a bounded volume hierarchy, a tree-like data structure which decreases the number of ray-intersection tests performed. Since each ray traced by the algorithm is independent, this algorithm also lends itself relatively well towards being parallelized (although not as well as rasterization). Even given some amount of parallelization, this algorithm requires a massive amount of computational power to produce a converged image. An example of a noisy non-converged image and converged image is shown in Figure 2.2. Reducing the noise to an acceptable level causes the computational cost to exceed what could be done in real-time on general parallelization hardware. Some variations of ray tracing, such as
Figure 2.2: A non-converged (left) and converged (right) image of the Sponza scene, rendered using the Blender Cycles renderer.

path tracing, do their best to limit the noise; but thus far have been unable to do so in real-time contexts.

For non-real time settings, ray tracing is one of the go-to methods for producing high quality photorealistic renderings. Its ability to simulate physical phenomena such as light dispersal through camera lenses, refraction through dielectrics, and indirect light bounces makes it perfectly suited for animated film renderers and VFX applications where photorealism is desirable. Blender’s Cycles renderer, Houdini, Pixar’s Renderman, and most other commercial rendering and animation software is built on the ray tracing/path tracing algorithm.
Table 2.2: Pros and cons of the ray tracing algorithm

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can simulate many realistic phenomena natively (shadows, dielectrics, etc.), therefore producing more photorealistic images</td>
<td>Very computationally intensive</td>
</tr>
<tr>
<td></td>
<td>More difficult to parallelize than rasterization</td>
</tr>
</tbody>
</table>

2.2 Vulkan

Real-time computer graphics applications, both 2D and 3D, rely on a rich ecosystem of APIs and libraries to function. Undoubtedly, the most important of these APIs is the graphics API, which allows application designers to interface with graphics processing units. From the late 90s to the mid 2010s, the most popular real-time graphics APIs were OpenGL and DirectX (or Direct3D). Both of these allowed for software developers to read/write to GPU memory and execute programs known as shaders on the GPU. In 2007, with the release of the parallel computing platform CUDA, developers began to see the GPU as more of a general purpose parallelization device rather than one used solely for graphics. Graphics developers of course wanted to make use of this too, so both OpenGL and DirectX included support for general parallelization in 2012 and 2008, respectively. They did this through the inclusion of programs known as compute shaders. By the mid 2010s, many computing disciplines had turned towards GPUs as a primary computing device. The increased use by largely performance-oriented developers made it clear that the abstraction offered by originally graphics-only APIs like OpenGL and DirectX 11 was too debilitating to API performance to be suitable long-term.
The Khronos Group, developer of OpenGL, and Microsoft, developer of DirectX, responded to this desire for more performance by creating two low-level modern graphics APIs: Vulkan (2016) and DirectX 12 (2015). The two APIs share a similar philosophy of having minimal amounts of abstraction over the existing GPU driver, enabling developers to fine tune performance for their specific use cases. Practically speaking, the main difference between these APIs is Vulkan has cross-platform support (Linux, Windows, Android, Nintendo Switch, etc.) whereas DirectX 12 is limited to Windows and Xbox systems primarily.

Vulkan, as a low-level graphics and compute API, provides a much more fine-grained layer of abstraction over modern GPUs than previous higher-level APIs did. Developers who use Vulkan are able to have high levels of control over GPU memory management, writing and sending commands for the GPU to execute, GPU/GPU synchronization, and GPU/CPU synchronization. Although by no means a comprehensive list, core Vulkan terminology is outlined below:

1. Pipeline: A set of shaders to be run on some input data. These are split into three categories: graphics pipelines (traditional graphics pipeline stages), compute pipelines (specifies a single compute shader), and ray tracing pipelines (specifies a set of shaders used for ray tracing).

2. Render Pass: For graphics pipelines only, an associated render pass is required to specify what images (color buffer, depth buffer, stencil buffer, etc.) are drawn to by different shader stages.

3. Command Buffer: A buffer containing commands to be executed on the GPU. These buffers are generally rerecorded each frame, containing commands such as binding pipelines, binding data (geometry, textures, etc.), drawing objects to a texture, and dispatching compute shaders.
4. Descriptor Sets: An object that specifies what GPU data a shader will have access to when executing. For example, if a shader needs access to a list of lights in the scene, a set pointing to a buffer of lights will be bound to the shader before execution.

5. Image View: A layer of abstraction on top of raw image data. This contains information such as the number of channels and format of the image, and is used when transitioning between image formats and interpreting the image in shaders.

6. Swap Chain: An abstraction between user code and the window used for presentation. Contains the raw images that will be displayed to the window and presentation mode information (immediate, vsync, etc.).

2.3 Real-Time Ray Tracing

The ability to render images with ray-traced levels of quality in real-time has been a dream of graphics developers for decades. Realism and detail in interactive 3D applications would see extreme improvements if it were feasible to trace a non-trivial number of rays each frame. For much of the ray tracing algorithm’s lifespan, there was little push towards making it real-time capable. The computation cost of doing so was too far out of reach for computing devices at the time. But in recent years, there have been major advances in both hardware and ray-tracing-related algorithms that have allowed for some level of raytracing to become part of real-time graphics developers’ toolkits.

The first major stride towards real-time ray tracing becoming possible was when consumer GPUs included support for generalized parallelization in the early 2010s. Using this technology, developers began experimenting with adapting CPU ray tracing
techniques to a parallelized environment. For example, ways to efficiently parallelize
the construction of bounded volume hierarchies and optimize ray triangle intersections
on the GPU were created at this time. These advancements resulted in massive
performance increases for renderers with access to GPUs, but were not enough to
permit ray traced rendering in consumer real-time graphics applications. Two big
hurdles still needed to be overcome to break into real-time:

1. Ray-triangle intersection calculations are slow, and millions of these occur each
   frame.

2. GPUs are much quicker when each thread of execution is doing the same thing,
   and when ray tracing, threads are very likely to diverge from one another.

The first big step towards solving one of these problems was Nvidia’s development
of their 2000 series of graphics cards, released in 2018. These cards, dubbed “RTX”
GPUs, were the first to include hardware acceleration for ray tracing calculations.
Specifically, these GPUs included dedicated ray tracing cores (RT cores) that added
hardware support for performing ray-triangle intersection calculations. This signif-
icantly lessened one of ray tracing’s biggest computational bottlenecks. Alongside
this, Nvidia provided built-in functionality for parallelized bounded volume hierarchy
construction, as well as a programming model depicted in Figure 2.3, both of which
make development of optimized GPU ray tracers easier. This new technology allowed
for up to 10 billion rays to be cast per second (10 GRays/s) with the RTX 2080ti, a
number far beyond what was possible previously in consumer GPU hardware.
Despite 10 GRays/s seeming like a massive number, in practice, this is too little to perform traditional ray/path tracing algorithms in a real-time setting. With current hardware, a more reasonable budget is four traces per pixel per frame, or 500 million rays per second at 60 frames per second. This 20x difference is the result of two factors. First of all, the GPU is not only doing ray tracing; other tasks like drawing UI, rasterization, collision detection, and more can all occur on the GPU as well. These all cut into the time spent tracing rays and limit performance in real-world settings. The other limiting factor is the geometric complexity of the scene. As the number of triangles in a scene grows, this, in turn, increases the time spent on ray-triangle intersection calculations. In many photorealistic applications that would look towards RTX ray tracing to improve image quality, the scene complexity is too high to permit the maximum rate of 10 GRays/s from occurring.

As a result of these constraints, a new family of “low rays per pixel” ray tracing algorithms have been developed for real-time applications. At the moment, the most common variation of this type of algorithm is selectively using the ray budget for reflections, shadows, ambient occlusion; things rasterization is typically bad at. Then
those results are blended with a rasterized image to create a more detailed image than either rasterization or ray tracing could produce independently. The alternative to this is ray tracing the entire image, but in games with typical levels of geometric and lighting complexity, this approach results in very noisy renders. This has spurred research into efficient denoising algorithms, where after the ray tracing is complete, a denoising stage occurs before the image is presented to the application user. An example of this is presented in Figure 2.4. There exist methods outside of these two general categories, some of which avoid an explicit denoising step, though most consumer applications with RTX support use one of these approaches.

![Figure 2.4: A render before and after a denoising step [24].](image)

### 2.4 Physically Based Rendering

Many 3D applications strive to achieve some level of photorealism. Photorealism is a combination of many factors (i.e. the arrangement of objects in a scene, object colors), though the most significant aspect is generally the way light interacts with
the scene. Because of this, in almost all rendering contexts, lighting is of paramount
importance for achieving the desired level of realism. Since near the inception of
computer graphics, methods have been developed for modeling light and materials in
ways that approximate reality. This style of approach is known today as “physically
based rendering” (PBR; popularized in 2014 by a book of the same name [27]), as it
takes into account both the real-world properties of materials and behavior of light
to produce photorealistic renderings.

2.4.1 Materials

When designing a 3D model for use with a physically based rendering system, there are
many factors outside of geometry to take into account. Specifically, having a detailed
representation of each model’s material to access when rendering is crucial. This
allows the rendering engine to consider more factors when light comes into contact
with the objects in a scene, resulting in more realistic light colors and bounces. There
are countless combinations of physical properties one could require in a PBR renderer,
so an overview of common properties is given below (note that some of these properties
overlap with one another, and would not be used in conjunction):

1. Base Color: The color of a material.

2. Diffuse Color: The base color, but only for diffuse light reflection.

3. Specular Color: The base color, but only for specular light reflection.

4. Metallic: Specifies if the given material should behave in a diffuse or dielectric
   manner.

5. Roughness: Controls the balance of diffuse and specular light reflection.
6. Glossiness: The inverse of roughness, also controls the balance of diffuse and specular light reflection.

7. Subsurface: A metric of how diffuse light is reflected.

8. Anisotropic: Defines how a material behaves at different viewing directions (i.e. velvet).

9. Sheen: A secondary component that helps define behavior at different viewing directions.

10. Clearcoat: Specifies the intensity of a secondary specular layer over the base material.

11. Clearcoat gloss: How glossy the aforementioned clearcoat layer appears.

Frequently, PBR parameters are used in conjunction with microfacet models. These models approximate the surface of a material as a collection of microscopic bumps of varying heights and orientations called microfacets. The orientation of an object’s microfacets is described through a statistical distribution centered around the object’s surface normal at a given point. For instance, a perfect reflector would have all of its microfacets pointing along the surface normal, whereas a perfect diffuser would have its microfacets scattered uniformly around the surface normal’s hemisphere. A graphic depiction of these microfacets can be found below in Figure 2.5.
2.4.2 Direct & Indirect Illumination

In order to model light in a realistic manner, graphics developers first consider how light scatters in the physical world. Although it does not model all light phenomena, the rendering equation presented in Figure 2.6 is the source of most light approximation algorithms.

\[
L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_{\Omega} f_r(x, \omega_i, \omega_o, \lambda, t)L_i(x, \omega_i, \lambda, t)(\omega_i \cdot n)d\omega_i
\]

Figure 2.6: The rendering equation [19]. Descriptions of each variable can be found in the appendix.

This equation states that the light at a given point is equal to a function of the sum of all light incident on the point from the hemisphere around it, plus any light that point may be emitting itself. The function that takes the incident light and calculates how much light is scattered in the direction of the viewer (or any direction for that matter) is known as the bidirectional reflectance distribution function, or BRDF. The BRDF is one component of the rendering equation for which there are many approximations.
Depending on the material model used and the desired visual style of the render, there are a huge number of BRDFs one could select from. In many circumstances, multiple BRDFs are even used concurrently for different objects within the same scene. A basic outline of solving the rendering equation is presented below:

**Algorithm 3:** Pseudocode for solving the rendering equation.

**Result:** The color being reflected in a specific direction from a point.

```plaintext
outColor = light being emitted from the point;

for the unit hemisphere centered around the point’s surface normal do
    outColor += BRDF of the incoming light;
end

return outColor;
```

Unfortunately, computationally solving the integral in the rendering equation is very difficult in almost all non-trivial circumstances. So, most PBR lighting algorithms attempt to approximate this integral with as little error as possible. Some approximations simplify the integral by splitting off a term for only light coming directly from emitters as in Figure 2.7. This is done by replacing the $L_i$ term (all incident radiance) with $L_e$ (direct emission only).

$$L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_{\Omega} f_r(x, \omega_i, \omega_o, \lambda, t) L_e(r(x, \omega_i), -\omega_i)(\omega_i \cdot n)d\omega_i$$

**Figure 2.7:** The rendering equation for direct lighting, summing over the unit hemisphere. Descriptions of each variable can be found in the appendix.

Direct lighting is the light reflecting off a point coming directly out of a light emitter. Since the light has only bounced once (that being off the point being looked at), this is sometimes referred to as single bounce lighting. In most circumstances, the vast majority of light visible in a scene is direct light. This is due to light losing a large portion of its luminosity upon successive bounces, making the impact of two, three,
four, etc. bounces much less than the first. Many real-time applications only concern themselves with direct lighting for this reason, as a believable level of realism can be achieved with a single bounce alone. Because this term in the rendering equation only adds up light coming from direct emitters, the integral can be reexpressed as summing over all light emitters in the scene, demonstrated in Figure 2.8.

\[
L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_A f_r(x, \omega_i, \omega_o, \lambda, t)L_i(x, \omega_i, \lambda, t)(\omega_i \cdot n)dA
\]

Figure 2.8: The rendering equation for direct lighting, summing over light emitters. Descriptions of each variable can be found in the appendix.

Indirect lighting is all the light in the scene that has bounced more than once. Although this term can have little influence depending on the circumstance, at times it is absolutely crucial in obtaining a photorealistic render. Specifically, indirect lighting allows objects to realistically reflect their surroundings, enables dielectrics (i.e., glass) to have proper transparency, and simulates color bleeding between objects close in proximity. This term is much more difficult to compute than direct lighting, as you need to sample all types of geometry - not just emissive geometry as in the direct lighting term. Achieving a good looking estimate of this term takes a significant amount of time, and therefore is rarely done in real-time applications outside of single bounce mirrors or glass.
Figure 2.9: One render including only direct lighting, and another including both direct and indirect [29].
3.1 RTX Algorithms

Recent work in real-time ray tracing has been split between using ray tracing in combination with rasterization and exclusively using ray tracing. Because RTX enabled graphics cards are relatively new and RTRT capable consoles are just beginning to be adopted, real-time 3D application developers have mainly used ray tracing for enthusiast-level additions rather than core features. Therefore, most industry examples favor the hybrid rasterization/ray tracing approach, as it is much easier to integrate with preexisting rendering engines (although there are a few notable exceptions).

Hybrid ray tracing approaches have had excellent graphical results in both research and industry applications. SEED, Electronic Arts’ research division, developed a hybrid renderer for their prototype application PICA PICA [14]. They utilized the rasterization, compute, and RT cores of the GPU to render the portion of the scene they were individually best suited to, and then blended the results. Enscape GmbH, a developer of a renderer used primarily for architectural visualization, published information on a hybrid renderer where most of the rays were traced in screen space, only utilizing the spatial data structure when necessary [15]. Likely the biggest industry examples of hybrid rendering systems are in AAA games such as Battlefield 5 [8], Cyberpunk 2077 [12], and Call of Duty: Black Ops Cold War [11]. These extremely popular titles provide RTX support for capable devices and use RT cores for applications such as shadowing, ambient occlusion, and reflections.
Figure 3.1: The various components that contribute to a render in SEED’s PICA PICA [14].

The exclusive use of ray tracing to render scenes introduces challenges not present in hybrid renders. The most glaring of these challenges is the tradeoff between compute time and noise, as the limited amount of time for a real-time render prevents single-frame image convergence in most circumstances. To combat this, researchers have focused on efficient techniques for denoising during a real-time application’s sequential frames. A large number of light emitters, one common source of noise, has been the focus of recent work in [23] and [16]. The development of more general denoisers, such as the one presented in [28], has also been advancing quickly. The real-time graphics industry has only produced a few RTRT-only applications: Minecraft RTX [13] and Quake 2 RTX [10]. These two titles are visually stunning, but both sit in specific graphical niches that permit exclusive use of ray tracing. Minecraft is a voxel-based game, so developers were able to heavily optimize their RTRT approach far more than would have been possible for a general 3D setting. Quake 2 RTX is simply a matter of scene complexity. The game uses the same assets as the original 1997 release, which by today’s standard are incredibly low-poly. Therefore, the cost of tracing a ray is reduced significantly, permitting real-time performance.
3.2 ReSTIR

Introduced in [16], Reservoir-based Spatio-Temporal Importance Resampling, or ReSTIR, is a ray tracing algorithm for calculating direct lighting designed for real-time use. ReSTIR is an interesting departure from other RTRT rendering approaches, as it does not require an explicit denoising step like most other current algorithms. The algorithm specializes in handling scenes with a large number of light emitters, with [16] demonstrating real-time performance in scenes with upwards of 20000 triangle lights. The key insight made by the designers of ReSTIR was to use spatial and temporal data to determine the most likely lights illuminating a given point, calculated using the filtering of light scattering probabilities. This results in a render that converges far more rapidly than traditional methods, handles camera movement, and supports varying levels of photorealism. The implementation of ReSTIR referenced in the original paper was done in Falcor, a DirectX 12-based graphics prototyping engine developed by Nvidia.
Figure 3.3: An image of the zeroday scene, rendered using the ReSTIR implementation from [16].

3.2.1 Reservoirs

ReSTIR’s core approach relies on two statistical techniques to sample light sources: resampled importance sampling (RIS) [31] and weighted reservoir sampling (WRS) [18].

In order to handle a large number of light sources while maintaining real-time frame rates, ReSTIR needs to selectively choose the most influential light sources for a given point rather than calculating the influence of them all individually. The term “importance sampling” refers to this exact process; instead of choosing lights at random to illuminate a point, lights that are more significant (or more ‘important’) in determining the point’s color are chosen. There exist many different importance sampling techniques, such as standard Monte Carlo importance sampling [26] and multiple-importance sampling [32]. Resampled importance sampling’s approach is to generate a collection of candidate samples from an easy-to-sample distribution (i.e.
the emissivity of light sources), and then select one of the candidates driven by a desired probability distribution function (i.e. the PDF of the rendering equation).

Weighted reservoir sampling is a process for sampling from a stream of input candidates, where each candidate has an associated weight. With only a single pass over the input candidate stream, it selects a sample with probability equal to the ratio of its weight and the sum of all candidate weights.

In [16], they combine these two techniques into a method known as streaming resampled importance sampling (streaming RIS). Weighted reservoir sampling is used to extend RIS’s sampling strategy into one that processes sequentially generated elements in a stream. Because elements are processed in a stream-like fashion, very little data is associated with the current sampling state. Algorithm 4 contains a class definition of the sample state, referred to in [16] as a reservoir.

**Algorithm 4:** The class definition of a reservoir, presented in [16]

```plaintext
class Reservoir

    y ← 0 // The output sample
    w_sum ← 0 // The sum of weights
    M ← 0 // The number of samples seen so far

function update(x_i, w_i)

    w_sum ← w_sum + w_i;
    M ← M + 1;
    if rand() < w_i/w_sum then
        y ← x_i;
```

24
Given the above class definition, Algorithm 5 contains the pseudocode for streaming resampled importance sampling.

**Algorithm 5**: Pseudocode for streaming resampled importance sampling, presented in [16]

<table>
<thead>
<tr>
<th>Result: A reservoir containing the sample state of the input pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>function</strong> <code>RIS(Pixel \text{q})</code></td>
</tr>
<tr>
<td>Reservoir ( r );</td>
</tr>
<tr>
<td><strong>for</strong> ( i \leftarrow 1 ) <strong>to</strong> ( M ) <strong>do</strong></td>
</tr>
<tr>
<td>generate ( x_i \sim p );</td>
</tr>
<tr>
<td>( r.update(x_i; \hat{p}_q(x_i)/p(x_i)) );</td>
</tr>
<tr>
<td><strong>end</strong></td>
</tr>
<tr>
<td>( r.W = \frac{1}{\hat{p}<em>q(r.y)} \cdot \left( \frac{1}{r.M} \cdot r.w</em>{sum} \right) );</td>
</tr>
<tr>
<td>return ( r );</td>
</tr>
</tbody>
</table>

In practice, this sampling technique is applied on a per-pixel level. For each pixel in the render, a reservoir is maintained. This reservoir samples some number of lights each frame, say \( M \), based on the emissivity of the light sources. It then selects \( N \) of the candidate samples, driven by the rendering equation’s PDF, and uses them to shade the pixel. Since each reservoir contains the sampling state of an entire stream of candidate samples, the combination of data from two reservoirs is enough to simulate the reservoir built from the concatenation of their two initial source streams. The algorithm for combining multiple reservoirs is presented in Algorithm 6. The process of combining multiple reservoirs is used to reduce noise, as more samples per
reservoir corresponds to a quicker rate of convergence. Two approaches to reservoir combination are presented in [16], temporal reuse and spatial reuse.

**Algorithm 6:** Psuedocode for combining multiple reservoirs, presented in [16]

```python
function combineReservoirs(Pixel q, Reservoir r_1, ..., Reservoir r_k)

Reservoir s;

foreach r ∈ {r_1, ..., r_k} do
    s.update(r.y, \( \hat{p}_q(r.y) \cdot r.W \cdot r.M \))

s.M ← \( r_1.M + r_2.M + \cdots + r_k.M \);

s.W = \( \frac{1}{\hat{p}_q(s.y)} \cdot \left( \frac{1}{s.M} \cdot s.w_{sum} \right) \);

return s;
```

### 3.2.2 Temporal Reuse

In RTRT applications, an extremely common technique to reduce noise is utilizing data from previous frames in the current frame’s render. The most naive way to accomplish this is accumulating samples each frame, and blending them with the new samples in the current frame. This is typically done on a pixel by pixel basis. This works well in applications with a still camera, but in interactive 3D applications generally a lot of camera movement occurs. In order to handle this, determining which (if any) pixel the geometry of interest occupied in the previous frame is necessary. Because most real-time applications use linear projection and view matrices, saving the previous frame’s matrices allows for backprojection into the previous frame’s pixel space.

ReSTIR adapts this into a simple temporal reuse algorithm that drastically increases render quality. For every pixel, backproject the geometry into the previous frame’s
pixel space. If the geometry existed in a valid pixel, combine that pixel’s reservoir with the current pixel’s reservoir. An example of this is presented in Figure 3.4. In order to prevent unbounded growth of previous frames’ reservoirs, [16] suggests clamping the weight of the previous frame’s reservoir to 20 times the current frame’s weight.

![Figure 3.4: An example of temporal backprojection.](image)

3.2.3 Spatial Reuse

In the vast majority of renders, spatial locality has a strong correlation with the influence of light sources. If a pixel is primarily illuminated by a specific light emitter, it is likely nearby pixels are also illuminated by the same light. ReSTIR exploits this by combining reservoirs of nearby pixels each frame, thus increasing the overall sample count per pixel. Bitterli, et. al [16] specifies some default parameters to consider for spatial reservoir reuse:

- Select five neighbors.
- Default search radius of 30 pixels.
- Compare camera depth of samples, reject if greater than 10% difference.
• Compare surface normals, reject if greater than 25 degree difference.

This specific variant of spatial reuse is biased, as it ignores checking if the light samples from neighboring pixels are visible from the current pixel. To correct this for unbiased spatial reuse, simply check that the neighbors’ reservoir samples are visible from the current pixel.

Figure 3.5: An example of temporal and spatial reuse in the Subway scene, from [16].
3.2.4 The Algorithm

With RIS, temporal reuse, and spatial reuse, pseudocode for ReSTIR can be written:

**Algorithm 7: RIS with spatiotemporal reuse, presented in [16]**

**Input**: Previous frame’s reservoirs

**Output**: Current frame’s reservoirs

**function reservoirReuse(prevFrameReservoirs)**

- reservoirs ← new Array[ImageSize];
  - // Generate initial candidates
  - foreach pixel q ∈ Image do
    - reservoirs[q].W ← RIS(q);
  - // Evaluate visibility for initial candidates
  - foreach pixel q ∈ Image do
    - if shadowed(reservoirs[q].y) then
      - reservoirs[q].W ← 0;
  - // Temporal reuse
  - foreach pixel q ∈ Image do
    - q′ ← pickTemporalNeighbor(q);
    - reservoirs[q] ← combineReservoirs(q, reservoirs[q],
      prevFrameReservoirs[q′]);
  - // Spatial reuse
  - for iteration i ← 1 to n do
    - foreach pixel q ∈ Image do
      - Q ← pickSpatialNeighbor(q);
      - R ← { reservoirs[q′] | q′ ∈ Q};
      - reservoirs[q] ← combineReservoirs(q,reservoirs[q],R);
    - foreach pixel q ∈ Image do
      - Image[q] ← shadePixel(reservoirs[q],q);
  - return reservoirs;
3.3 BRDFs

3.3.1 Metallic-Roughness

GL Transmission Format (glTF) files are a popular way of storing 3D models for real-time applications. glTF files store geometry, material, and texture data in either binary or unpacked data structures. The default material model supported by this file type is the metallic-roughness material, with an associated metallic-roughness BRDF defined in the specification [2]. Along with supporting the metallic-roughness parameters, glTF files also include support for normal, emissive, and occlusion maps by default.

The process of creating models for use with this material system is outlined in [21]. The three main material properties taken into account are geometry’s base color, how metallic the geometry is, and the geometry’s roughness. When creating a model for use with this material system, typically this is done through the creation of two textures: the base color and metallic-roughness maps. An example of this is presented in Figure 3.6.

The actual interpretation of these values is done in the metallic-roughness BRDF. The function is a blend of metallic and dielectric components, and uses a microfacet-based approach for its specular light reflection known as the GGX microfacet distribution [34]. In Figure 3.7, the full BRDF is summarized.

3.3.2 Disney “Principled”

For real-time applications, it is common to forgo a large number of material parameters for less data-overhead and a simpler BRDF. In renderers striving for higher levels
Figure 3.6: An example of metallic-roughness textures, from [21].

Figure 3.7: A summary of the metallic-roughness BRDF [2].
of realism however, such as film or VFX renderers, it is necessary to consider many
different types of material properties to accurately calculate lighting. To achieve de-
sired visual effects, it is even common to use multiple BRDFs and materialing systems
within the same scene. In 2012, Disney released information on their Disney “Prin-
cipled” BRDF, which was used for “virtually every material but hair” in Wreck-It
Ralph [17]. This BRDF is an extremely powerful tool, as it can model a myriad
of materials while keeping the number of material parameters low (one color and
10 scalars). Although this is slower than simpler BRDFs (i.e. metallic-roughness),
it does provide artists with huge amounts of creative control for a relatively small
penalty. Figure 3.8 depicts the effects of tuning various BRDF parameters.

Since the popularization of the Disney BRDF, there has been lots of work done to
adapt and extend it in different contexts. In [30], the details of importance sampling
the Disney BRDF are derived. Work done by [7] extended the BRDF into a bidirec-
tional scattering distribution function (BSDF), which models light transmission as
well as reflection. Developers of Blender adapted the original Disney BRDF into a
BSDF of their own, known as the “Blender Principled BSDF” [4]. Since this material
model is so flexible and simple, it is also compatible with software such as Pixar’s
Renderman and the Unreal Engine.

The use of more complex material models with ready-to-ship 3D asset formats like
glTF does present some challenges. Natively, glTF does not have support for all
the parameters utilized by the Disney BRDF. This could be solved on the applica-
tion programmer’s end by having an intermediate loading step that includes the
additional parameters, although this requires a fair amount of development over-
head. Perhaps the cleaner solution is using extensions to the glTF file format (i.e.
KHR_materials_clearcoat, KHR_materials_sheen [20]), which add support for addi-
tional parameters beyond the metallic-roughness parameters.
Figure 3.8: A chart depicting the influence of the Disney BRDF parameters [17].
4.1 Vulkan Abstraction

The Vulkan API is designed to provide minimal abstraction over the GPU driver, enabling developers to optimize performance for their specific application. In order to optimize for real-time rendering, a family of singleton “manager” classes were developed in this thesis. Each manager provides a layer of abstraction over a specific subset of Vulkan features useful in real-time rendering contexts. Beyond just providing abstraction, dividing core Vulkan API calls across multiple classes makes the codebase far more readable, as the verbosity of the Vulkan C and C++ APIs can lead to lengthy and hard-to-decipher code. Note that of the two versions of the Vulkan API, the C++-style API provided in vulkan.hpp was used in this project to further increase readability and type-safety.

Each of the rendering styles supported by this engine are built on top of the abstraction provided by these managers. A diagram depicting the overall system architecture is presented in Figure 4.1.

4.1.1 Buffers

One of Vulkan’s most desirable features is the level of control it provides over GPU memory management. The core Vulkan library includes the ability to allocate heap and image memory on the GPU; leaving the management of said memory completely up to the user. For applications which use fairly standard GPU memory allocation
Figure 4.1: An overview of the renderer’s architecture. Blue tiles are explained further in subsequent figures.
patterns, such as this renderer, a common approach is using the Vulkan® Memory Allocator (VMA) library managed by AMD GPU Open. This library provides a built-in heap memory manager and defragmenter, making it far easier to allocate memory for many small buffers on the GPU. This library has had success in many industry applications, such as Google’s Filament rendering engine and a majority of Vulkan game titles on the PC [6].

The BufferManager class provides a wrapper around VMA functionality necessary for the real-time rendering applications explored in this thesis. The allocation and deletion of buffers and textures, the copying of data between different blocks of memory/memory types, and device memory address querying are all provided here. Much of the interaction with this manager happens at program startup, as most memory allocation occurs when loading geometry, textures, and initializing shader-related data. Any GPU allocations present at program shutdown are cleaned up through an RAII-style deallocation pattern. In Figure 4.2, a comprehensive look at the provided BufferManager functionality is presented.

4.1.2 Assets

The three types of assets supported in this thesis are glTF 2.0 files for model data, HDRI environment maps, and BRDF lookup tables (contained in .png files). The AssetManager class provides the core functionality for interacting with these files and managing their associated data. In the case of the HDRI environment maps and BRDF lookup tables, the management of these is simple enough to contain completely within the AssetManager class. The data contained within these files is uploaded to the GPU through interaction with the BufferManager, and can be requested for later use in rendering operations through AssetManager methods. Models are much more
Figure 4.2: A detailed look at the BufferManager singleton.

complex to manage however, and therefore are encapsulated in their own instances of the Model.h class.

Model loading for glTF files is managed by one of two libraries: the TinyGLTF library for models with precomputed tangents, and The Open-Asset-Importer-Lib (Assimp) for models without. The models are loaded in the AssetManager, and then passed to the Model constructor. Here, model data such as vertices, indices, materials, and textures are parsed and subsequently uploaded to the GPU. In addition to this, a “bottom level acceleration structure” (BLAS) is created on the GPU for later use with ray tracing operations. A detailed look at the AssetManager and its associated Models is provided in Figure 4.3.
Figure 4.3: A detailed look at the AssetManager singleton and the associated Model class.
4.1.3 Descriptors

Vulkan’s method of exposing data to shaders is through Descriptor Sets. Compared with other graphics APIs, descriptor sets offer developers a wealth of micro-level control over how shader resources are exposed. However, writing abstractions over this portion of the API is all but necessary to maintain a readable codebase. Here, the abstraction is provided in the DescriptorManager class.

In order to create a descriptor set for use with a shader, it is first necessary to encapsulate the data’s layout in a `vkDescriptorSetLayout` structure. Following this, room for the descriptor set needs to actually be allocated on the GPU. Finally, actual handles to the buffers/textures of interest can be written to the descriptor set. The DescriptorManager class handles two types of sets, mutable and immutable. Immutable sets are relatively easy to create and manage, so adding them is done manually when necessary. For mutable sets, the process is a bit more complicated. Since multiple frames can be in-flight on the GPU at the same time, a descriptor set for each possible frame in-flight is necessary to prevent race conditions involving descriptor set updating. Multiple template functions are provided in the DescriptorManager for creating, updating, and acquiring mutable descriptor sets of arbitrary data structures. The provided functionality for the DescriptorManager is presented in Figure 4.4.
4.1.4 Misc

The CommandManager, QueueManager, and PresentationManager only encapsulate a small amount of Vulkan abstraction, and are therefore relatively straightforward.

Vulkan command buffers are lists of commands to execute on the GPU. The CommandManager contains vkCommandPools that command buffers are allocated from, and allows for allocation of a few different command buffer types.

![Diagram of CommandManager singleton](image1)

Figure 4.5: A detailed look at the CommandManager singleton.
The QueueManager contains the queues through which commands are submitted to the GPU. Multiple queue families, such as queues specifically for data transfer, are available here. This manager contains small abstractions over queue submission processes, such as an option to force a CPU-side executing stall until the queue is finished executing on the GPU.

![QueueManager diagram](image)

**Figure 4.6: A detailed look at the QueueManager singleton.**

The purpose of the PresentationManager is twofold: it contains the GLFW windowing system used for actual presentation, and it manages the Vulkan swapchain used to interact with the windowing system. The most notable use of this class is requesting the current frame’s image and ID to allow for proper drawing and descriptor set binding.
4.1.5 Renderer

The main place all of the provided Vulkan abstractions are used is within the Renderer class. Here is where the specific rendering methods described in this thesis are implemented. These methods will be described in further detail in the three following sections. In addition to the rendering methods defined here, there are some necessary abstractions included here as well. Namely, the creation of assets related to pipelines and render passes occur here. Shader loading, the creation of graphics/compute/RT pipelines, render pass creation, framebuffer management, and CPU/GPU and GPU/GPU synchronization all occur here. These operations themselves rely on most of the manager classes to occur, so they sit alongside the actual rendering procedures in the class creation hierarchy. The abstractions provided within the Renderer are detailed in Figure 4.8.
4.2 PBR Rasterization

The first rendering method developed for this thesis was a physically-based rasterization renderer. Built almost entirely from the traditional graphics pipeline, this rendering style implements the metallic-roughness model detailed in the glTF 2.0 specification. In addition to this, a skybox and environment mapping are supported as well. Shadowing from a single light source is also supported, using a ray visibility test detailed in a further section. Because of the relative simplicity of CPU-side operations for this rendering technique, the vast majority of compute time is spent on the GPU in vertex and fragment shaders.

Most of the detail from this physically-based rendering approach is packaged into texture-based data structures. Specifically, the geometric data is augmented with normal, emissive, occlusion, metallic-roughness, and base color maps. Each of these
are interpreted in different ways to augment how the geometry interacts with or emits light; drastically improving render quality without increasing the number of triangles present in the mesh.

The base color map is interpreted in conjunction with the metallic-roughness map in a fragment shader to form the core of this rendering style. The metallic term controls if a material is a metal, dielectric, or some blend of the two. For metallic materials, light is modeled to reflect off the surface of the object (entirely specular), whereas for dielectric materials light partially enters the object and is absorbed and scattered (diffuse and specular). The roughness term controls how much a dielectric material scatters light before ejecting it from its surface.

**Figure 4.9:** A render depicting the effect of changing metallic and roughness values.

Adding detail to a model is done through the inclusion of the other three maps: occlusion, emissive, and normal. Normal mapping is an extremely common technique present in real-time applications where you read your surface normal from a texture
rather than by interpolating vertex normals. The normal map texture is typically stored in tangent space, so the core of this technique relies on transforming the value read from the texture into world space. This can be done using a perturbed normal method which almost entirely occurs in a fragment shader [1], however, in this application a traditional approach of constructing the transformation matrix in the vertex shader is implemented. Tangent and bitangent vertex values read or calculated from the input glTF model are interpolated to allow for smooth normal mapping transformations.

Integrating the emissive and ambient occlusion maps is far easier, as neither requires any spatial transformations to occur. If an occlusion map contains a non-zero value, the final output color is simply scaled by that value to simulate some level of occlusion. Similarly, if the emissive map contains a non-zero value, it is added to the final output color. In combination, these five maps give the 3D asset creator a substantial amount of control over how objects will appear in this rendering style.

Figure 4.10: A render of the “damaged helmet” glTF model utilizing all five material maps.
Looking at a single frame, there is very little asynchronicity in terms of execution. A frame begins by first initializing a command buffer and recording a request to build a top level acceleration structure (TLAS). This acceleration structure contains all the scene geometry and is used to later shadow the scene. Following this, a render pass is recorded including two separate pipelines: a skybox pipeline and a metallic-roughness pipeline. The skybox pipeline is very simple, as it only includes a single descriptor set binding for the skybox texture and an associated draw command for the cube it is projected on. The second pipeline is a fair bit more complex. Because it utilizes the TLAS constructed earlier in the frame, a memory dependency must be recorded into the buffer before this pipeline executes. Within the pipeline itself, each model in the scene records a draw command for every instantiation of itself in the scene. Along with these draw commands, associated descriptor sets including texture and position data are bound as well. Lastly, the entire command buffer is submitted to the GPU for execution.

On the GPU-side of this process, almost all the compute time is spent in the metallic-roughness pipeline’s fragment shader. This is where nearly all of the texture lookups and math processes occur, as lighting calculations take place here. Lighting is limited to two sources: a directional light source (the sun) and an environment map. The metallic-roughness BRDF’s specular and diffuse components are first calculated for the directional light source and later blended with the contribution from the environment map. Before shadowing, the contributions from the occlusion and emissive maps are included. The final step is the shadow calculation process, which traces a ray from each fragment to the directional light source to determine occlusion. The final product of this process is an image including metallic-roughness PBR shaded objects with shadows and an HDRI skybox.
Figure 4.11: An overview of the metallic-roughness physically-based rendering style.

4.2.1 Preprocessing

To prevent the size of assets for the application from bloating to an unreasonable degree, some preprocessing is done on the input HDRI environment map before real-time rendering begins. First, HDRI environment maps are generally packaged as an equirectangular projection; to allow for ease of use, this is transformed into a cubemap on the GPU using a simple fragment shader. Once this cubemap has been generated, two additional processes are run to create a filtered and convoluted version of the cubemap. The convolution process essentially blurs the cubemap, giving a good estimation of the indirect diffuse lighting emitting from the environment. Filtering the cubemap creates an estimate of the indirect specular component of the environment, containing highlights where light emission is particularly strong. In the filtered cubemap, a continuously blurrier version of the filtered environment is stored at each mip-level. This is done to simulate different levels of material roughness without needing to re-importance sample the cubemap in real-time.
4.2.2 Ray Query Shadows

A supported Vulkan extension with relatively little literature written about it is the VK_KHR_ray_query device extension. The ray query extension allows use of the RT cores for real-time ray tracing within the traditional graphics pipeline. For example, within a fragment shader any number of rays could be generated and shot around the scene. These rays would traverse an acceleration structure provided as a descriptor set and make full use of RT core hardware acceleration. Unfortunately, ray queries provide relatively little information about the intersection when compared with a full ray tracing pipeline, but have enough detail to perform processes like shadowing and ambient occlusion.

Traditionally, shadows are calculated through “shadow mapping”, a rasterization process in which additional renders of the scene are performed from the perspective of light sources. Because additional renders and spatial transformations are required,
the overhead for this method can be incredibly high when dealing with multiple light
sources or area lights.

In order to both test the performance of ray queries and increase the simplicity of
my shadowing solution, I implemented dynamic shadowing using fragment shader
ray queries. The method itself is extremely simple; for each fragment, shoot a ray
towards the directional light source. If the ray hits anything, then the fragment is
shadowed — otherwise it remains fully lit. The simplicity of this method cannot be
understated, as a full shadowing solution can be achieved in merely 10-15 lines of
shader code. In order to avoid rays intersecting with the fragment they are generated
from, they are moved slightly along the geometric surface normal before intersection
calculations begin.

Figure 4.13: A render of the Sponza scene with the full PBR rasterization
rendering style.
4.3 Hybrid ReSTIR

This rendering approach implements the ReSTIR real-time ray tracing algorithm described in [16] with support for the Disney Principled BRDF. Similar to the PBR rasterization method, this approach uses large portions of the glTF 2.0 model specification. Additional values outside those described in the specification are also provided for use with the Disney BRDF, namely scalars controlling anisotropic, sheen, clearcoat, and subsurface effects. This technique supports many thousands of triangle light emitters for direct lighting, including accurate shadowing.

Algorithmically, this has been broken into three main sections: a rasterization step, a ray tracing step, and a compute shader step. First, within a rasterization pipeline a G-Buffer is rendered containing geometry present in the view frustum. Following this, light samples are generated in a ray tracing pipeline and temporal reuse occurs with the previous frame. In a compute shading pipeline, these samples are reused spatially and final lighting is calculated for the frame.

![Diagram of Hybrid ReSTIR rendering style](Figure 4.14)

**Figure 4.14:** An overview of the Hybrid ReSTIR rendering style.

For a single frame, the instructions are recorded into a command buffer as follows:

1. A command to build a TLAS for the current scene is recorded.
2. A render pass and associated rasterization pipeline are bound for rendering the G-Buffer. Within this pipeline, the commands are identical to those used in the metallic-roughness PBR rendering style, with the caveat that no skybox render occurs.

3. The ray tracing pipeline is bound and executed, containing shaders for ray generation, ray intersection, and ray misses.

4. The compute shader is bound and one thread is dispatched for each pixel in the render.

Each of these stages is memory-dependent on one another, and therefore has execution barriers separating them. Each stage will be covered in more detail in the following subsections.

The final product of this process is an image including Disney BRDF (direct lighting) shaded surfaces and emissive triangles.

4.3.1 G-Buffer Generation

A geometry buffer (or G-Buffer) is a collection of textures containing different information about the geometry visible by the camera. Information such as positions, normals, and material data are generally stored in a G-Buffer.

For the G-Buffer generation step, much of the machinery is identical to the metallic-roughness PBR rasterization rendering method. Instead of drawing with the metallic-roughness fragment shader, a specialized G-Buffer shader is used instead. The G-Buffer shader writes out data into multiple render targets (MRTs), with each of the targets containing data in specific formats to use in future steps. In particular, there are eight render targets drawn in this step: a position texture, normal texture,
emissive texture, depth buffer, and four material parameter textures. The material parameter textures include the arguments for the Disney BRDF, with scalar values each occupying a single RGBA channel for a given texture.

![Renders of some G-Buffer targets for the Sponza scene.](image)

**Figure 4.15:** RENDERS OF SOME G-BUFFER TARGETS FOR THE SPONZA SCENE.

### 4.3.2 Ray Generation & Temporal Reuse

Following the generation of the G-Buffer, a ray tracing pipeline is executed. A ray tracing pipeline is made up of a collection of shaders, each handling different aspects of the ray tracing algorithm. This pipeline contains a ray generation shader, ray intersection shader, and ray miss shader. The miss and intersection shaders are extremely simple, as they just return a boolean representing if a given ray hit anything or not (although they can be far more complex). The ray generation shader is therefore where most of the computation in this pipeline occurs.

For every pixel in the output render, a thread of the ray generation shader is executed. Each thread loads G-Buffer data from its assigned pixel and then proceeds with the ReSTIR algorithm. A reservoir is generated through the resampled importance sampling routine, cycling through 32 candidate light sources as suggested by [16]. Then, a ray is generated with its origin at the position loaded from the G-Buffer, and
its target being a randomly sampled position on the selected light source. If the ray intersects any geometry between the origin and target, the weight of the reservoir is set to zero. This accounts for the geometry being occluded (or shadowed), as light was unable to travel to the geometry of interest uninterrupted.

Temporal reuse occurs next, combining the current reservoir for the pixel with its temporal neighbor’s reservoir from the previous frame. Using temporal backprojection, the temporal neighbor is located and its associated reservoir is loaded. The backprojection routine performs a clipping process to ensure the geometry was actually visible in the view frustum on the previous frame. If a previous reservoir was found, its weight is clamped to 20 times the current frame’s weight to prevent unbounded growth as specified in [16]. Finally, the new reservoirs are stored in two textures: a reservoir and reservoir sample texture. The reservoir texture contains the actual reservoir sample and weight, whereas the sample texture contains the barycentric coordinates of the light source sample.

The above process can be extended to support multiple reservoirs per-pixel by simply repeating it multiple times. In the implementation developed for this thesis, four reservoirs per pixel are supported.
4.3.3 Spatial Reuse

The final step before calculating lighting is performing spatial reuse between the reservoirs. A compute shader is dispatched in warps of size 64 (for 8x8 blocks of pixels) to run both the spatial reuse and lighting calculations. Just as in the ray generation shader, G-Buffer data is loaded to gather information about the geometry of interest. In addition, data about the current pixel’s reservoirs are loaded as well.

The spatial reuse function of this compute shader first selects five neighbor pixels in a 30 pixel radius of our target pixel. Then, depth information and surface normals are compared to validate that these are similar-enough reservoirs to warrant combining. The constants used for comparison are the same as in [16]; no more than a 25 degree surface normal or 10% camera depth difference. Multiple passes of spatial reuse can be performed as well to increase pixel samples, with this implementation opting for two passes.
After the combination of reservoirs, actual lighting calculations can occur. Using the data in the G-Buffer in combination with the light sample selected by the reservoir, all the necessary data for the direct lighting rendering equation is present. The Disney BRDF can be evaluated with the input parameters, giving an output color for each pixel in the output image. If multiple reservoirs are used, the rendering equation is evaluated for each of the light samples and averaged for a final result.

Figure 4.17: A render of the Sponza scene without (left) and with (right) spatial reuse. The effect is particularly notable around harsh shadow edges.

4.4 RT-Only ReSTIR

The final rendering method implemented for this thesis is a variation of the original ReSTIR algorithm from [16]. There are two main deviations from the original algorithm: geometric information is gathered through ray tracing instead of a G-Buffer, and indirect lighting is factored into the rendering equation. Architecturally, this simplifies the design presented in the hybrid ReSTIR rendering style. Now, gathering information about the geometry in the view frustum is done within the same
ray generation shader as the resampled importance sampling and temporal reuse. In addition to this change, a single bounce of indirect lighting is also sampled within the ray generation shader. This simplifies the rasterization $\rightarrow$ ray tracing $\rightarrow$ compute pipeline structure used for hybrid ReSTIR into a two step ray tracing $\rightarrow$ compute structure.

![RT-Only ReSTIR](image)

**Figure 4.18: An overview of the RT-Only ReSTIR rendering style.**

For a single frame, the instructions are recorded into a command buffer as follows:

1. A command to build a TLAS for the current scene is recorded.

2. The ray tracing pipeline is bound and executed, containing shaders for ray generation, ray intersection, and ray misses. The ray generation shader simulates light scattering out of the camera lens to determine visible geometry. This emulates G-Buffer generation, but allows for greater control over the type of camera perspective used.

3. The compute shader is bound and one thread is dispatched for each pixel in the render.
Each of these stages is memory-dependent on one another, and therefore has execution barriers separating them. Stages will be covered in more detail in the following subsections.

The final product of this process is an image including Disney BRDF (direct & one bounce indirect lighting) shaded surfaces and emissive triangles from the perspective of a fisheye lens.

4.4.1 Fisheye Lens

The largest visible benefit of using ray tracing instead of a G-Buffer for determining visible geometry is the simulation of different types of camera lenses. The traditional rasterization graphics pipeline supports linear projections, such as perspective or orthographic, but fails to support realistic camera models (generally non-linear). Realistic camera lens simulation involves tracing rays from the view plane out into the scene, passing through layers of camera lenses to determine final position/direction. For this rendering style, a fisheye lens (non-linear) is supported. Simulating a fisheye lens works well in real-time contexts, as it only requires tracing rays through a single lens.

In order to accurately simulate a fisheye lens, rays are scattered out of the view plane by converting pixels’ UV coordinates into polar coordinates. These rays are scattered uniformly out of the view plane by keeping the angular distance between them constant [22]. If the polar coordinate has a radius larger than one, then no ray is scattered for that pixel, as it would not be visible from the fisheye perspective. Different degrees of fisheye projection are also supported, with any value greater than zero and less than or equal to 180 supported.
Because of the non-linearity of fisheye lenses, performing temporal backprojection is much more difficult. There is no way to easily determine the pixel a piece of geometry occupied in the previous frame, as is possible with the linear perspective projection. To cope with this, the temporal reuse weight is clamped to 5x rather than 20x, and only reuses samples from the same pixel.

Figure 4.19: A render of the Sponza scene with the RT-Only ReSTIR 180° Fisheye Lens, without second bounce lighting.

4.4.2 Indirect Lighting

Only calculating direct lighting, as is done in the hybrid ReSTIR rendering style, commonly results in extremely dark scenes. To combat this, a single indirect lighting sample is also calculated in this approach. In the ray generation shader, a ray is traced in a random cosine-weighted direction. The ray returns information about the geometry hit, which is later used in the final shading process. This substantially increases the brightness in shadowed areas of the scene, simulating a realistic level of ambient light dispersal.
Unfortunately, the indirect lighting bounce is extremely noisy. As this indirect bounce does not use advanced sampling or reuse strategies, the noise it produces pollutes the more-converged direct lighting sampled regions. Addressing this in a physically realistic way would be very computationally costly, so a simple gaussian blur is applied to the indirect lighting instead. This vastly reduces the noise present, while maintaining a believable level of realism. Renders produced using this approach are a nice blend of the lighting present in the metallic-roughness PBR rasterization and the hybrid ReSTIR approach, as shadows are present but not incredibly harsh.

Figure 4.20: A render of the Sponza scene with the RT-Only ReSTIR 180° Fisheye Lens, including second bounce lighting with a gaussian blur.
Chapter 5

RESULTS

5.1 Renders

The figures presented in this section are exemplary of each rendering method’s capabilities. Four screenshots from each method are presented, with most focused on the Sponza and San Miguel scenes common for lighting tests. Each was rendered using a NVIDIA GeForce RTX 2070 Super in conjunction with an AMD Ryzen 3800x.

Figure 5.1 and Figure 5.2 illustrate the PBR rasterization method’s materializing system and environment mapped lighting. In order to make the environment lighting more apparent, RTX shadowing is disabled for these renders. Figure 5.3 and Figure 5.4 represent the full capabilities of the metallic-roughness rasterization approach, with the scene complexity provided by the Sponza and San Miguel models better representing industry real-time graphics applications. As these two renders make use of the entire PBR rasterization approach, RTX shadowing is of course enabled.
Figure 5.1: A metallic-roughness rasterization render of a suit of armor with a studio environment map, shadows disabled.

Figure 5.2: A metallic-roughness rasterization render of a damaged helmet with a park environment map, shadows disabled.
Figure 5.3: A metallic-roughness rasterization render of the Sponza scene with a park environment map, shadows enabled.

Figure 5.4: A metallic-roughness rasterization render of the San Miguel scene with a park environment map, shadows enabled.
Each of the renders utilizing the Hybrid ReSTIR method illustrate a different quality of the approach. Figure 5.5’s render of the Sponza scene contains advanced shadowing around the light coming between the curtains and pillars. Similarly, Figure 5.6 demonstrates advanced shadowing in Sponza from a lamp centered in the middle of the model, with the lamp shade obstructing much of the scene from the emitter. A byproduct of supporting an arbitrary number of light sources in real-time is some level of noise being present in the final render, which occurs in Figure 5.7. In Figure 5.8, a render of the San Miguel scene at dusk (illuminated by a distant torus) contains more advanced shadowing, similar to the Sponza sphere and lamp renders.

![Figure 5.5: A Hybrid ReSTIR render of the Sponza scene with 15360 triangle emitters.](image)

Figure 5.5: A Hybrid ReSTIR render of the Sponza scene with 15360 triangle emitters.
Figure 5.6: A Hybrid ReSTIR render of the Sponza scene with a lamp containing 15360 triangle emitters.

Figure 5.7: A Hybrid ReSTIR render of the Sponza scene with 46080 triangle emitters, divided equally between 3 different colored spheres.
The renders produced for the RT-Only ReSTIR method are very similar to those produced for Hybrid ReSTIR. The largest departure is a focus on largely shadowed areas, particularly in Figure 5.9 and Figure 5.12, as these are the areas most affected (brightened) by the inclusion of a single bounce of indirect light. The Sponza lamp scene used in the Hybrid ReSTIR renders is also used in Figure 5.10. Outside of the indirect lighting, the biggest difference introduced by this approach is the increased noise in Figure 5.11. Because spatial and temporal resampling is more difficult in a RT-Only context, more noise is present here than in the similar Figure 5.7 Hybrid ReSTIR render.
Figure 5.9: A 180° RT-Only ReSTIR render of the Sponza scene with 15360 triangle emitters and single-bounce indirect lighting.

Figure 5.10: A 180° RT-Only ReSTIR render of the Sponza scene with a lamp containing 15360 triangle emitters, including single-bounce indirect lighting.
Figure 5.11: A 180° RT-Only ReSTIR render of the Sponza scene with 46080 triangle emitters (divided equally between 3 different colored spheres), including single-bounce indirect lighting.

Figure 5.12: A 180° RT-Only ReSTIR render of the San Miguel scene lit by an orange torus containing 18432 emissive triangles, including single-bounce indirect lighting.
5.2 Performance

In regards to performance, the main consideration is average frametime. Many different factors could be altered here for interesting analysis, but for brevity’s sake, altering the number of triangles in the view frustum was selected. This simplifies what occurs CPU-side, as only one draw call for a mesh containing n triangles needs to occur each frame. For GPU-side operations, altering the number of input triangles should give a good representation of how the three rendering approaches scale relative to scene size.

The first plot, presented in Figure 5.13, shows a relatively linear increase when scaling up the number of triangles using the PBR rasterization method. When looking at loads under 10000 triangles drawn each frame, the frametime is essentially insignificant GPU-side (less than 1ms). With per-fragment RTX shadowing enabled, performance is expectedly slow when looking at large triangle-count scenes. Around 10,000,000 triangles per scene dips the average frames per second below 30, a common goal for real-time graphics applications.

Figure 5.14 and Figure 5.15 each show a relatively uniform frametime up until jumping from 10,000,000 to 100,000,000 triangles. Interestingly enough, although each is constant, their baseline frametimes differ a large amount. The Hybrid ReSTIR method always takes around 40ms (even with low triangle counts), whereas the RT-Only ReSTIR method only takes around 10ms while constant. This implies that the context switch between rasterization and ray tracing pipelines costs a large amount of time in the Hybrid approach, whereas when that computation is moved inside of the ray tracing pipeline that penalty is not incurred. Under 10,000,000 triangles, both methods achieve decent levels of real-time performance, with the RT-Only method providing more features at the cost of increased levels of noise.
Figure 5.13: A plot depicting the number of rasterized triangles vs. frametime using the PBR rasterization method.

Figure 5.14: A plot depicting the number of triangles in the scene vs. frametime using the Hybrid ReSTIR method.
Figure 5.15: A plot depicting the number of triangles in the scene vs. frametime using the RT-Only ReSTIR method.
Chapter 6

FUTURE WORK

6.1 Engine Improvements

Outside of the specific rendering methods implemented for this thesis, the actual rendering engine has lots of room for growth. Specifically, three major areas for improvement are descriptor set management, CPU-side parallelization, and implementing a dependency graph architecture.

The creation and management of descriptor sets is considered one of the more difficult aspects of the Vulkan API. The level of detail needed at runtime for each individual shader can lead to bloated and hard-to-manage codebases. This is without a doubt present in this thesis’ rendering engine, as a decent chunk of new code is necessary for each individual shader. Likely the cleanest solution available at the moment involves using the Khronos Group’s SPIRV-Cross library [3]. SPIRV-Cross allows parsing of the SPIRV shader binaries used by the renderer, giving runtime information about the data each shader expects to be available. With this information, the engine could support reading in shaders without any prior knowledge of their contents; therefore drastically decreasing the overhead for editing or introducing new shaders.

None of the three rendering methods implemented in this thesis are CPU-bound, so many CPU-side optimizations were not prioritized. A relatively simple inclusion that would have fantastic benefits would be parallelizing the model loading process. Currently, models are loaded in alphabetical order with no multithreading — an extremely slow process when dealing with high-detail models. This does not affect
runtime performance at all, but when loading in many complex models at once the startup time for the renderer becomes unbearable.

To further improve performance and usability, a clear target would be including support for a rendering dependency graph. This approach allows for management of an entire frame’s execution through a graph, in which different rendering/compute passes and execution/memory dependencies are specified. With modern graphics APIs like DirectX 12 and Vulkan, this is quickly becoming an industry-standard design pattern. Unreal Engine 4 even has core support for this architecture, with it being advertised as a way to easily perform full-frame optimizations [5].

6.2 PBR Rasterization Lighting

The metallic-roughness PBR rasterization method produces quality renders, but has a somewhat lackluster featureset. Only supporting environment and directional light sources severely limits the types of scenes that can be rendered, as it is near impossible to render within largely enclosed spaces. To combat this, developing support for more light sources of varying types would be required. Including support for some number of point light sources would be an easy addition that would drastically increase the number of scenes that this method would support.

Ray query shadowing, although a simple solution, without a doubt results in major performance bottlenecks in scenes with a large number of triangles. Changing the shading sample rate may have some effect on this performance hit, but the clear solution would be moving the shadowing into a separate ray tracing pipeline. Having a dedicated RT pipeline for shadowing is a solution other renders have explored with good levels of success, such as in SEED’s PICA PICA prototype game [14]. Having a
separate RT pass would also make it easier to include other effects, such as ray traced
ambient occlusion or reflections.

6.3 ReSTIR Features

Both the Hybrid and RT-Only ReSTIR implementations support the full feature set
described in [16], with the exception of the unbiased spatial reuse pass and alias
table for importance sampling light sources. The alias table would result in a better
light selection process, therefore increasing render convergence speed. Development
time for the alias table would be relatively short, and would likely result in better
render quality for little-to-no computation cost on a per-frame basis. Similarly, it
would take little development time to implement the unbiased spatial reuse pass.
The performance hit incurred would be significant, but the number of reservoirs used
at each pixel could be altered to reduce the penalty somewhat. Having the option
of using the unbiased version would allow for interesting performance and visual
comparisons between the two methods.

Outside of features described in the original ReSTIR paper, the biggest issue with
the RT-Only ReSTIR method is the inability to do accurate temporal backprojection.
Since the fisheye lens model is non-linear, simple methods for determining a position’s
reservoir in the previous frame are infeasible. Exploring ways to combat this would
be an interesting line of inquiry, as a massive amount of render quality when camera
movement is occurring is the result of backprojection.
BIBLIOGRAPHY


    https://github.com/KhronosGroup/gltf/tree/master/specification/2.0.


    shader/principled.html.

    US/ProgrammingAndScripting/Rendering/RenderDependencyGraph/.


[7] Extending the disney brdf to a bsdf with integrated subsurface scattering. 2015.


[9] Introduction to nvidia rtx and directx ray tracing.


Appendix A

VARIABLE DEFINITIONS

\[ L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_{\Omega} f_r(x, \omega_i, \omega_o, \lambda, t)L_i(x, \omega_i, \lambda, t)(\omega_i \cdot n) d\omega_i \]

where:
- \( L_o \) is the outgoing radiance
- \( x \) is the location
- \( \omega_o \) is the outgoing direction
- \( \lambda \) is the wavelength of light
- \( t \) is time
- \( L_e \) is the light emitted
- \( \Omega \) is the unit hemisphere centered around \( x \) (in the area form of the equation, this is replaced with the surface of all emitters \( A \))
- \( f_r \) is the BRDF
- \( L_i \) is incoming radiance
- \( \omega_i \) is the incoming direction
- \( n \) is the surface normal

Figure A.1: The rendering equation [19] with variable definitions. Note that the same definitions apply for the direct lighting and area rendering equations as well.