5-2024

Empowering Graphics: A Distributed Rendering Architecture for Inclusive Access to Modern GPU Capabilities

Taylor Anderson
Utah State University, taylor.anderson@usu.edu

EMPOWERING GRAPHICS: A DISTRIBUTED RENDERING ARCHITECTURE FOR INCLUSIVE ACCESS TO MODERN GPU CAPABILITIES

by

Taylor Anderson

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Mario Harper, Ph.D. Major Professor	Dean Mathias, Ph.D. Committee Member
Chad Mano, Ph.D. Committee Member	D. Richard Cutler, Ph.D. Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2024

ABSTRACT

Empowering Graphics: A Distributed Rendering Architecture for Inclusive Access to Modern GPU Capabilities

by

Taylor Anderson, Master of Science

Utah State University, 2024

Major Professor: Mario Harper, Ph.D.
Department: Computer Science

This thesis proposes the design and creation of a distributed rendering architecture to utilize one user's powerful GPU to allow others to experience a similar graphic fidelity with less powerful machines. This architecture will allow users to share unused computational power to render dense and computationally difficult scenes for other users whose GPU may be unable to handle the scene. By rendering in such a way, users will be able to benefit from the latest advancements in rendering, such as hardware ray-tracing or Nvidia's DLSS, even on graphics cards lacking native support for these features. This architecture differs from previous remote solutions, such as Google's Stadia, by relying on a small, local group of peers and being built into the rendering engine itself, rather than relying on a remote server farm.

To showcase the renderer's capabilities, the rendering engine and code are be fully built out and demonstrated on a graphically intensive, moderately complex scene built for higher-end hardware. Additionally, the latency and performance of the graphics stream will be evaluated, through observing the latency of input-to-visual results and performance on local networks. The rendering engine will support modern computer graphics techniques,

such as physically based rendering, to increase the complexity of the rendering and evaluate the performance as it would pertain to modern graphics technology stacks.

The importance of this thesis is in the pursuit of access for more people to modern graphics. With GPUs becoming increasingly expensive and many games requiring the newest hardware for the optimal experience, many individuals are inhibited from experiencing the full potential and experience of the game's design. This thesis facilitates hardware sharing among users, extending a similar graphics experience to a wider audience, while utilizing pre-existing hardware.

(46 pages)

PUBLIC ABSTRACT

Empowering Graphics: A Distributed Rendering Architecture for Inclusive Access to Modern GPU Capabilities

Taylor Anderson

Modern rendering software requires powerful GPUs with the latest hardware features in order to utilize all of the newest rendering techniques. Many users do not have access to this hardware, and rely on remote server farms or reduced performance to achieve usable results. In this thesis, the software is designed and created to allow for a user to share the resources of their computer with another, modeling a split-screen setup like was common in the past, but without requiring users to be in the same location.

By designing the software from the ground up to support this, instead of adding the capabilities to a finished product, the software can achieve higher performance than would be possible with each user rendering on their own machine. The software provides means to allow users to share their hardware capabilities with others, making high-quality graphics more freely available.

To my brothers and family who have helped support me.

ACKNOWLEDGMENTS

Thank you to Dr. Mario for his help and guidance in my academic journey.

Taylor Anderson

CONTENTS

Page

ABSTRACT . . . iii

PUBLIC ABSTRACT . . . v

ACKNOWLEDGMENTS . . . vii

LIST OF FIGURES . . . x

ACRONYMS . . . xi

1 Introduction . . . 1

2 BASICS OF RENDERING . . . 2
- 2.1 Light . . . 2
- 2.2 Camera . . . 3
- 2.3 Scene and Models . . . 3
- 2.4 Rasterization . . . 5
- 2.5 Depth Buffer . . . 5
- 2.6 Materials . . . 6
- 2.7 Shader . . . 6
- 2.8 Render Pipeline . . . 7

3 PHYSICALLY BASED RENDERING . . . 8
- 3.1 Introduction . . . 8
- 3.2 Roughness, Smoothness, and Microfacets . . . 8
- 3.3 Conservation of Energy . . . 9
- 3.4 Reflectance Equation . . . 10
  - 3.4.1 Radiometry and Radiance . . . 10
  - 3.4.2 Solid Angle . . . 11
  - 3.4.3 Radiant Flux . . . 11
  - 3.4.4 Radiant Intensity . . . 11
  - 3.4.5 Final Radiance Equation . . . 11
- 3.5 BRDF . . . 12
  - 3.5.1 Distribution Function . . . 13
  - 3.5.2 Geometry Function . . . 14
  - 3.5.3 Fresnel Equation . . . 14
- 3.6 Cook-Torrance Reflectance . . . 15

4 VULKAN 16
4.1 Introduction 16
4.2 Vulkan Workflow 16
4.2.1 Initialization 16
4.2.2 Per Frame 18
4.2.3 Cleanup 19

5 STREAMING 20
5.1 General Streaming 20
5.2 Host 21
5.3 Client 21
5.4 Render Loop Changes 22

6 RESULTS 24
6.1 Introduction 24
6.2 Host Performance 24
6.2.1 Frame Rate Impact 24
6.3 Remote Performance 25
6.3.1 Frame Rate Impact 25
6.3.2 Latency 26
6.3.3 Compression Effects 26
6.4 Discussion 27
6.5 Future Work 27

REFERENCES 29

APPENDICES 31
A Rendered Outputs 32
A.1 Encoding Quality Effects 32

CURRICULUM VITAE 35

LIST OF FIGURES

Figure Page

2.1 Our perception of color comes from the light that is reflected off of objects. 2

2.2 A representation of a pinhole camera. The light from an object passes through the small hole on the front and lands on the back of the box. . . . 3

2.3 Side by side image showing a model of an avocado, as well as a wireframe render of the same avocado. The vertices and edges that make up the model can be easily seen in this manner. 4

2.4 The full render pipeline. All stages are shown, although this renderer only modifies the geometry and pixel shaders. 7

3.1 A microscopic view of a surface reveals the small irregularities in the surface, with small deformations causing light to bounce in erratic ways. The roughness attribute models this behavior. 9

5.1 The relationship between host and client is shown. The raw pixel data from the host's GPU is encoded to JPG format before transfer to the client, where it will be decoded back into the raw pixels. 21

A.1 Example render with a quality of 90. With this quality setting for the JPG encoding, visual artifacts are generally unseen, although perceptible in the gradient. 33

A.2 Example render with a quality of 75. Decreasing the quality to this level starts to create more apparent visual artifacts in the transmitted image. . . 33

A.3 Example render with a quality of 50. The visual artifacts are even more visible, especially in the gradient. 34

A.4 Example render with a quality of 25. At this quality, the visual artifacts are obvious, and the solid black area has clear errors near the border of the dark and lighter areas. 34

ACRONYMS

RGB	Red, Green, Blue
glTF	GL Transmission Format
PBR	Physically Based Rendering
BRDF	Bidirectional Reflective Distribution Function
IOR	Indices of Refraction
FPS	Frames per Second

CHAPTER 1

Introduction

This thesis presents the development of a renderer with the capability to render for both a local and remote user simultaneously. The core focus of this research is to demonstrate the architecture and implementation of a renderer with these remote capabilities built into its core functionality. Additionally, the background behind modern rendering practices is discussed.

Modern rendering software uses physically based methods to achieve physically accurate scenery, as well as allowing artists to create models that are easily transferable between different rendering software, and look correct in any lighting condition. Physically based rendering is based on approximations of light, and following the laws of physics, specifically that the outgoing light from an object should never be greater than the light going into it. Traditional rendering approaches utilizing Phong shading allow for this to occur, which can lead to unpredictable outcomes in various conditions.

Real-time rendering for a remote user is not a new concept, however the approach usually utilizes expensive GPU server farms, such as Google's Stadia platform, which necessitates separate applications for each user. The implemented method in this renderer allows the remote user to take advantage of the memory already loaded into the GPU from rendering the host user, which allows for a smaller overall impact on the host machine. This method allows for the sharing of GPU resources between individuals easily, with hardware already owned by the users. Additionally, a lightweight, custom TCP protocol is implemented to facilitate the communication between client and host. The code for this thesis is available at https://github.com/tganderson0/crow-engine .

CHAPTER 2

BASICS OF RENDERING

2.1 Light

The appearance of an object is determined based on the environment's lighting mixing with the material properties of an object. As light hits an object's surface, a portion of the light is absorbed, and the remainder reflected, which is shown in figure 2.1. Our observation of color stems from this process. For example, a red apple appears as it does because it absorbs all wavelengths of light except those which correspond to the color red, which are then reflected toward our eyes, enabling us to perceive its color. The perceived color of an object is a combination of the light's color reflecting off of it, illustrated by the difference between the warm light of a candle and the cool light of a computer monitor, and the object's material properties, which influences the inherent red color of the apple. As the complex model of light is hard to encapsulate in software, we typically represent the colors using the Red, Green, and Blue (RGB) model, with values typically ranging from 0 to 1. This model can capture the broad spectrum of colors observable in our environment, although it is not perfect.

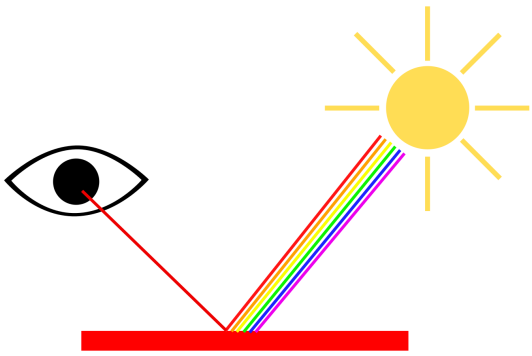

Fig. 2.1: Our perception of color comes from the light that is reflected off of objects.

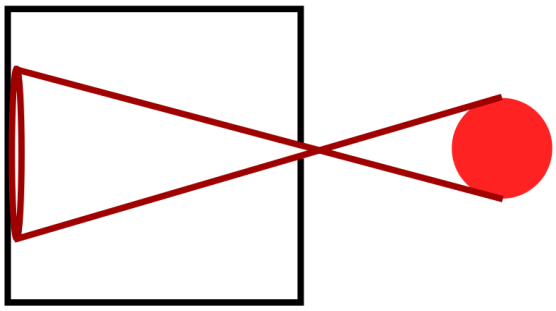

Fig. 2.2: A representation of a pinhole camera. The light from an object passes through the small hole on the front and lands on the back of the box.

2.2 Camera

The camera functions as a surrogate eye for the observer, capturing the scene's light. In practice this most often employs the concept of a pinhole camera [1] to simulate the vision of the user. The pinhole camera is essentially a simplified representation, consisting of a box with a small aperture on one side. This aperture allows light from the scene to penetrate the box and project onto the surface, as shown in figure 2.2.

A major difference between the physical pinhole camera and the virtual model that we use, is in the positioning of the image plane. In the virtual pinhole camera, the image plane is placed in front of the pinhole, rather than at the back. This ensures that the captured image is oriented correctly, rather than being inverted as it naturally would in a real pinhole camera.

2.3 Scene and Models

Models are typically constructed from a collection of points, known as vertices, and the connections between those vertices, which are called edges. By combining the vertices and edges, polygons can be formed, with triangles being the most prevalent due to their simplicity and efficiency in the rasterization process [2,3]. These elements come together to build more complex models which make up what we see after rendering. Figure 2.3 displays the edges and vertices for a model of an avocado. While the vertices and edges describe the geometric shape of an object, the model can additionally describe the visual attributes,

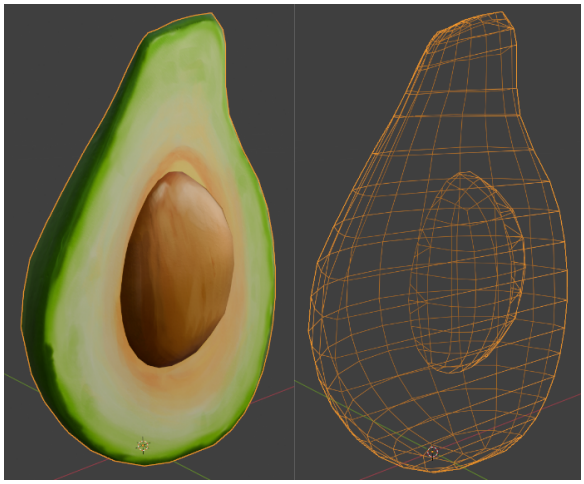

Fig. 2.3: Side by side image showing a model of an avocado, as well as a wireframe render of the same avocado. The vertices and edges that make up the model can be easily seen in this manner.

we can define additional material properties, such as color, roughness, and metallicness, and other other properties we want to use in the rendering process. The scene is all of the objects in the environment, their position, and orientation. In this manner, the vertices, edges, and polygons can be compared to the cells of an organism, making up the model, or organs, with the full scene being the organism itself. Models are typically constructed from a collection of points, known as vertices, and the connections between those vertices, which are called edges. By combining the vertices and edges, polygons can be formed, with triangles being the most prevalent due to their simplicity and efficiency in the rasterization process [2,3]. These elements come together to build more complex models which make up what we see after rendering. Figure 2.3 displays the edges and vertices for a model of an avocado. While the vertices and edges describe the geometric shape of an object, the model can additionally describe the visual attributes, we can define additional material properties, such as color, roughness, and metallicness, and other other properties we want to use in the rendering process. The scene is all of the objects in the environment, their position, and orientation. In this manner, the vertices, edges, and polygons can be compared to the cells of an organism, making up the model, or organs, with the full scene being the organism itself.

This renderer uses the glTF 2.0 (GL Transmission Format) specification for models, which defines the format in which the model is saved. There are many other formats can be saved in, such as OBJ, FBX, USD, BLEND, many CAD file types, and many more. glTF 2.0 was chosen as it is a royalty-free specification [4], can describe full scenes, as well as necessary material data for the specific rendering workflow of the engine, specifically with describing physically based materials.

2.4 Rasterization

After projecting the vertices of a model from 3D to a 2D viewing plane, the renderer must find all of the corresponding pixels for each of the primitives. This process is known as rasterization. One important not is that the color for each pixel is not decided at this stage, and is instead processed by the step after the rasterizer.

As the rasterization is performed at the hardware level, we cannot modify the process with shaders, but we can specify small changes, such as using multi-sampling or other anti-aliasing techniques. This allows the renderer to take multiple samples per pixel, instead of only one.

2.5 Depth Buffer

When following a rasterization workflow, objects may not be rendered in order from back to front. This can lead to issues where objects that should be behind another object might be rendered as if it were in front, since it is drawn later. One method of handling this is by using a depth buffer, or z-buffer. The depth buffer is the same dimensions as the final rendered image, however it only has one channel, as we don't need all three to represent depth. As objects are rendered, after multiplying the projection matrix, the depth of the object is tracked in the depth buffer. If the current pixel's depth has not been set yet, then the depth is recorded for that location, and the corresponding color pixel is drawn. If the current pixel's depth has been set already, then the depth of the new object is compared to the old. If the depth of the new object is less than the previous, the new depth is recorded and the object is drawn, while if the depth is greater than the previous, that pixel (for the

current object) is skipped, and we continue on. In this way, objects can be rendered in any order while producing results that make sense.

Transparent and translucent objects must be handled separately, by rendering after all other objects. This allows objects like glass to show what is behind it, while still being hidden by objects in front of it.

2.6 Materials

A material describes the physical properties of an object. In traditional rendering, a material consists of 3 parts, an ambient, diffuse, and specular property. This lighting model is also known as the Phong shading model [5], and while can still be used, isn't as commonly found anymore. A more modern approach to materials and lighting is utilizing Physically Based Rendering (PBR), which will be covered in the next chapter. The physically based materials generally have three main attributes, an albedo, roughness, and metallicness, although more can be used for more complex rendering.

2.7 Shader

A shader is a program written for the GPU that describes a vertex or pixel, utilizing information passed in as input. Typically, there are 2 main shaders in a renderer, the vertex shader, and a fragment or pixel shader. A renderer may have any number of vertex and fragment shader combinations, but both are required. Various languages can be used to write shaders, however you are usually limited to what your graphics library allows. As this renderer uses Vulkan, the shader language that Vulkan uses is SPIR-V. SPIR-V is a binary form, and can be compiled from any shading language, like GLSL and HLSL. This renderer utilizes the GLSL shading language for all shader code, and is built to SPIR-V before running.

A special type of shader is a compute shader, which allows arbitrary calculations to be performed on the GPU, similar to CUDA, but using SPIR-V like the vertex and fragment shaders. As the compute shaders are not built into the standard render pipeline, they have to be handled slightly differently in how you pass and retrieve data from them.

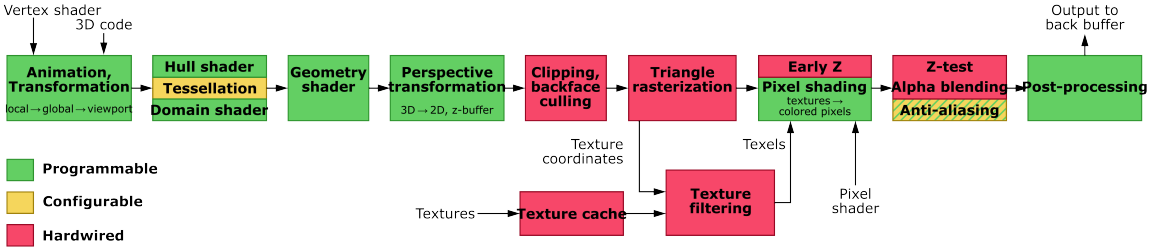

Fig. 2.4: The full render pipeline. All stages are shown, although this renderer only modifies the geometry and pixel shaders.

2.8 Render Pipeline

The render pipeline is comprised of the steps that takes the model and produces an image that can be displayed on the screen. In the most simple and common form, an object's data, in the form of vertex data, is passed to the GPU and ran on a vertex shader. The output of the vertex shader is used by the rasterizer, which gives necessary values for each visible pixel. This output is given to the fragment shader, also commonly called the pixel shader, which ultimately gives the output image. Variations on this basic render pipeline exist, especially when performing post-processing on the image. Additionally, this only refers to the rasterization pipeline, and a ray traced pipeline follows a different model.

There are a number of additional steps in the render pipeline, but most have more specialized use cases and aren't strictly necessary to modify for a renderer. The full pipeline is shown in figure 2.4.

CHAPTER 3
PHYSICALLY BASED RENDERING

3.1 Introduction

In the previous chapter, the most basic forms of rendering were used, namely the ambient, diffuse, and specular workflow, or Phong shading. However, materials in the real world are difficult to describe using those 3 parameters, and can be adversely affected by scene lighting changes, and is generally not as realistic as it can be. Physically based rendering (PBR), are still an approximation of reality, but is focused on being founded on physical parameters, and describing light in a more realistic way. PBR naturally allows objects to look correct in different lighting conditions by utilizing these physical parameters. One important thing to note, is that while PBR attempts to more correctly model the properties of light, it does not require the scene to be hyper-realistic. Walt Disney Pictures' highly stylized movies utilize PBR principles to have physically based lighting, while still having a highly stylized art style [6].

3.2 Roughness, Smoothness, and Microfacets

At its core, PBR is built on the idea that surfaces are a series of microscopic facets, and the roughness of these facets will affect the scattering and reflection of the light hitting the surface [7–9]. The effect on incoming light can be seen in figure 3.1. The surface is described with these microfacets, but as the microfacets are smaller than a pixel, the surface is instead defined statistically. The higher the variation of the surface normal, the greater roughness the material has. The variation is measured by the amount of microfacets that are aligned with the halfway vector, which is the normal between the light and view (camera) vectors. The formula for the halfway vector can be seen in equation 3.1. In this equation, the halfway vector is given as a function of the light and view vectors. We can make the assumption

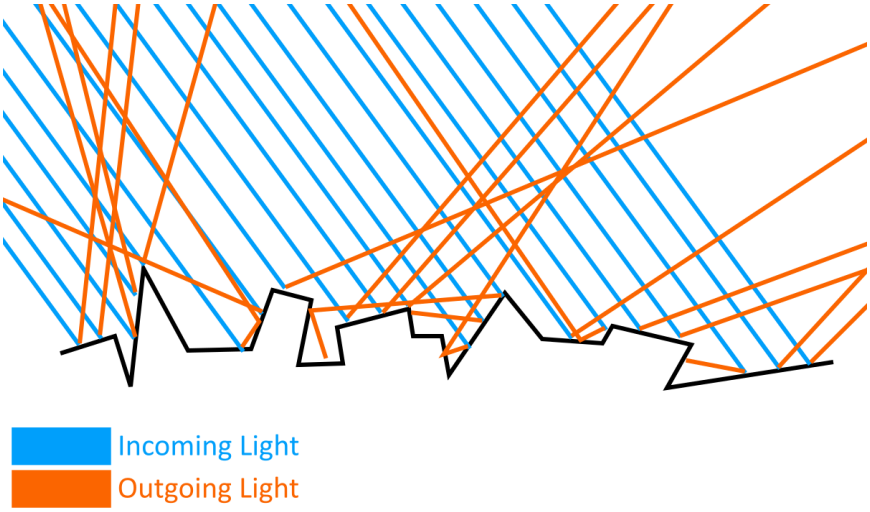

Fig. 3.1: A microscopic view of a surface reveals the small irregularities in the surface, with small deformations causing light to bounce in erratic ways. The roughness attribute models this behavior.

that a large number of microfacets are lit up by our incoming light, and as such can treat the aggregate behavior as the actual behavior for a given pixel.

$$h = \frac{l + v}{||l + v||} \tag{3.1}$$

3.3 Conservation of Energy

Another major principle of PBR is the conservation of energy. Our previous lighting models can break the laws of physics by outputting more light, or energy, from the surface than was put into it. PBR on the other hand, does not allow for outgoing light to be greater than incoming. PBR lighting is split into 2 stages, diffuse and specular. The diffuse is the light that is refracted when it hits a surface, while the specular is the light that is reflected. A simplifying assumption that is made however, is that all the diffuse lighting is simply absorbed at the area it hit the object, rather than being able to scatter, bounce, and exit the object at a different area. A small amount of this light is emitted as the object's diffuse color. There are techniques that allow for a more accurate simulation of the light, which is called subsurface scattering. If you cover a flashlight with your finger, you can see what

Name	Symbol	Unit
radiant flux	Φ	watt (W)
radiant intensity	I	W / m^2
radiance	L	W / (m^2sr)
irradiance	E	W / (m^2)

Table 3.1: A table of various radiometric quantities, which are methods of measuring and quantifying light.

subsurface scattering looks like in the real world. This properly models the behavior of the light bouncing within an object and reemerging at a potentially different area.

The important reason to split the light into diffuse and specular is to ensure the total outgoing light is never greater than the incoming. In order to do this, we first calculate the percentage of outgoing reflected light, then subtract that from the total in order to get our maximum diffuse lighting. This ensures that added together, we will never exceed the light that comes in.

3.4 Reflectance Equation

The full reflectance equation can be seen in equation 3.2. The equation applies generally to rendering [10,11], so as we break down each part so that it can be more easily understood, we will replace the simplified version with our actual equations.

$$L_o(p, w_o) = \int_\Omega f_r(p, w_i, w_o) L_i(p, w_i) n \cdot w_i dw_i \tag{3.2}$$

3.4.1 Radiometry and Radiance

L in the reflectance equation denotes the radiance for a light. It quantifies the strength of light coming from a direction, and is the combination of a few physical properties. In order to understand radiance, it is important to learn a few radiometry terms. Table 3.1 shows the relationship between each of the terms, as well as the units which it represents.

3.4.2 Solid Angle

The solid angle, w gives the area of a shape projected onto the unit sphere. It is an extension of the concept of an angle into a three-dimensional space.

3.4.3 Radiant Flux

The radiant flux, Φ is the flow of radiant energy over time, measured in Watts. The energy contained in light in the real world can be represented as a function using all the different wavelengths of light. Each color of light comes from the wavelength of the light. Radiant flux is the total area of the energy function. This isn't very easy to model in software however, so it's represented using RGB values, with some loss of information.

3.4.4 Radiant Intensity

The radiant intensity gives the amount of radiant flux per solid angle, which is the strength of a light source over the solid angle. The equation for radiant flux can be seen in equation 3.3.

$$I = \frac{d\Phi}{dw} \tag{3.3}$$

3.4.5 Final Radiance Equation

Using the three measures, we can then get the full equation for the full radiance, or L. It is the total energy in an area A over the solid angle w of a light with radiant intensity Φ. The full equation can be seen in equation 3.4. Radiance is the most important to us, as it is the value that physical cameras measure when taking photographs.

$$L = \frac{d^2\Phi}{dAdwcos(\theta)} \tag{3.4}$$

Going back to the reflectance equation 3.2, we know now that $L(p, w_i)$ represents the radiance of a point p and an incoming angle w_i, which is a direction vector. Because the light reflects more the more perpendicular it is to the surface, we get the $n * w_i$.

In the reflectance formula, we are calculating the sum of reflected radiance $L(p, w_o)$ of the point p in the direction w_o which is the direction to the camera. The full formula is the for the irradiance, which is the sum of all incoming radiance within a hemisphere Ω. The hemisphere is aligned with the surface's normal n. For each light that can cast onto the point p in the hemisphere, we calculate the radiance and add it, which is the integral part of the reflectance formula. A more accurate version can take into account the environment lighting as well, but the most simple version only uses the lights. All of the incoming light directions w_i as summed, scaled by f_r and gives the sum of all reflected light L_o in the camera direction.

3.5 BRDF

A bidirectional reflective distribution function (BRDF) gives the approximation of a surface's material properties using the microfacet theory as described before. The BRDF needs to respect energy conservation in order to be physically accurate, although for online rendering, approximate conservation is adequate. A BRDF is not required to be physically accurate, but we would not be doing PBR in that case. There are a number of different BRDFs that have been found in research [5, 7, 9, 12–15], but Cook-Torrance is the most commonly used one. Most BRDF models are built on top of Disney's Principled BRDF, such as Unreal Engine 4's BRDF they use [16].

The BRDF takes the light direction w_i the view direction w_o, the surface normal n, and an extra parameter a that represents the roughness of the surface, the average microfacet alignment as mentioned earlier. This BRDF approximates how much a given light ray contributes to the reflected light of an object taking into account the material properties. If the surface is a mirror, the BRDF would return 0 for all incoming light except for the singular ray that reflects at the same angle as w_o, where it would return 1.

A BRDF takes into account both the reflective (specular) and refractive (diffuse) material properties, and the equation can be seen in equation 3.5.

$$f_r = k_d * f_{lambert} + k_s * f_{cook-torrance} \tag{3.5}$$

In the equation, k_d refers to the ratio of incoming light which is refracted, or the diffuse part, while k_s is the ratio that is reflected. It should be noted that $k_d + k_s = 1$. $f_{lambert}$ refers to Lambertian diffuse, which uses the equation shown in equation 3.6.

$$f_{lambert} = \frac{c}{\pi} \tag{3.6}$$

In this equation, c is the albedo (surface color) of the object, at the given point. We divide by π to normalize the diffuse light since the full reflectance model is scaled by π. The diffuse section of the BRDF model can be swapped out for more physically accurate models, but Lambertian diffuse is fast and gives a pretty good approximation, which is a trade-off required for real-time rendering. Other models are more commonly used when dealing with offline rendering.

The specular section of the BRDF is much more complicated than the diffuse, which is shown in equation 3.7.

$$f_{cook-torrance} = \frac{DFG}{4(w_o \cdot n)(w_i \cdot n)} \tag{3.7}$$

This specular function is made up of four parts, with D, F, and G referring to three functions, and the denominator being a normalization factor. D, F, and G each represent a part of the surface's reflective properties. D is short for the Distribution function, F is short for the Fresnel equation, and G for the geometry function. Just like the diffuse function, each of the D, F, and G functions can be swapped out for more or less physically accurate versions, depending on rendering constraints.

3.5.1 Distribution Function

The distribution function approximates the alignment of the microfacets with the halfway vector, using the roughness of the surface. Its job in the reflectance function is to represent the effect of microfacet alignment with overall lighting result. We use the Trowbridge-Reitz GGX for the distribution function, which is the same as Unreal Engine

4. The equation for Trowbridge-Reitz GGX can be seen in equation 3.8.

$$NDF_{GGXTR}(m, h, \alpha) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2} \tag{3.8}$$

To break this function down, h is the halfway vector between the surface normal and the viewing angle, n is the surface normal, and α is the roughness measure we discussed previously. Roughness is squared as found in [6] using roughness squared for α resulted in more linear change in roughness, which simplifies asset creation.

3.5.2 Geometry Function

The geometry function is the function that approximates the self-shadowing of a microfacet. As seen in figure 3.1, some sections in the microfacet model are not lit by a light source, as other microfacets block light from reaching the surface, and thus cannot reflect in those areas. Just like the distribution function, this function can be chosen depending on accuracy and speed trade-offs. The function is shown in equation 3.9. For this, we're using Schlick-GGX, which is a combination of GGX and Schlick-Beckmann.

$$G_{SchlickGGX}(n, v, k) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k} \tag{3.9}$$

Once again, n is the normal of the point on the object, while v is the view vector (of the camera). The parameter k is a mapping from roughness to a different value depending if we are using a direct light or environment lighting.

3.5.3 Fresnel Equation

The Fresnel equation approximates the ratio of reflection at different viewing angles. A material has a base reflectivity, but any angle greater than perpendicular and the reflectivity will be higher. This maxes out at parallel with the surface, where the material will fully reflect light. Augustin-Jean Fresnel first described the phenomenon, and the Fresnel equation allows us to know how much light is reflected and refracted. The reason this

equation can be chosen differently however, is due to the fact that the Fresnel equations are computationally intense. Instead, an approximation is used, specifically the Fresnel-Schlick approximation. The equation can be seen in equation 3.10.

$$F_{Schlick}(h, vF_0) = F_0 + (1 - F_0)(1 - (h \cdot v))^5 \tag{3.10}$$

F_0 is the base reflectivity of the surface. This is calculated with the indices of refraction (IOR). A problem we will face with the Fresnel-Schlick function is that it is only defined for non-metal surfaces. For metal surfaces, the base reflectivity using IOR doesn't work, so we would have to use a different Fresnel approximation for all the metals in a scene. This is difficult to do, especially for objects that are partial metallic and partial non-metallic (like painted metal, with parts of the paint chipping off). Instead, we can pre-compute the F_0 using the surface color and a 'metalness' parameter. Most non-metal surfaces have a IOR of approximately 0.04, so we can linearly interpolate between a value of 0.04 and the surface color, based on the 'metalness' factor. We use surface color, as metallic surfaces have tinted IOR, while non-metal surfaces do not. This allows for the metal surfaces to have the tinting effect, while non-metals can remain as is.

3.6 Cook-Torrance Reflectance

Now that we have broken down each part of the equation, we can write the full equation that is used, rather than the generic PBR reflectance one. This can be seen in equation 3.11.

$$L_o(p, w_o) = \int_\Omega (k_d \frac{c}{\pi} + \frac{DFG}{4(w_o \cdot n)(w_i \cdot n)}) L_i(p, w_i) n \cdot w_i dw_i \tag{3.11}$$

I have still represented the DFG section with their abbreviated forms, to make it easier to read, as the fully written out version does not fit on the page.

CHAPTER 4
VULKAN

4.1 Introduction

In the previous chapters, we covered the fundamentals of rendering, the render pipeline, as well as the principles of physically based rendering. With that knowledge, we can move to actually starting to render. Vulkan is a cross-platform API for 3D graphics. The Vulkan API is an open standard created by the Khronos Group, as the successor of the OpenGL standard. Vulkan makes some key changes on OpenGL, focused on increasing performance and better CPU and GPU utilization. Vulkan additionally resembles more closely how a GPU works by working as a lower-level API than OpenGL. By using this lower-level API, Vulkan provides tighter control over what happens on the GPU. An additional advantage of Vulkan is that it is native on multiple operating systems, unlike Direct3D 12.

Common terminology and data types are shown in table 4.1. This table is essential for understanding the engine source code.

4.2 Vulkan Workflow

4.2.1 Initialization

As we are doing online rendering, the very first thing we need to do initialize our window we display in, which for this we use SDL2. Using SDL2 allows us to abtract away OS specific code, instead of requiring Windows or Linux specific setup. After that, we can move to creating our VkInstance, using that to create both a VkPhysicalDevice, as well as a VkDevice, with the extensions we want enabled on the GPU. Once we have those, we can create our swapchain.

As Vulkan does not handle synchronization itself, we need to create all of our fences and semaphores for rendering. Once we've done that, we can create our descriptor sets,

Term	Description
VkInstance	Stores the application state. Unlike OpenGL, there is no global state, so all state is stored here. This stores crucial information that is passed to the GPU.
VkPhysicalDevice	A reference to the GPU. Using Vulkan we can query the capabilities of the device for features like hardware raytracing, which is helpful for features that are not necessarily required by Vulkan. Additionally the physical device allows us to select which GPU to use in the case of multiple.
VkDevice	Rather than a reference to the GPU like the VkPhysicalDevice, the VkDevice is a reference to the actual GPU driver, and is the method to interact with our chosen GPU. All the extensions and features we want to utilize are enabled upon creation of the VkDevice.
VkImage	A handle to a multidimensional array of data. The most common use for VkImage is for images and textures. VkImage is usually not used directly, and requires an additional piece to descibe how to read and utilize the data contained within.
VkImageView	Contains metadata necessary for the GPU to utilize a VkImage. Shaders cannot directly utilize the images, and the image view describes the format of the data. This is especially apparent for cubemap textures, which actually is 6 images, but is just a blob of data in the VkImage.
Swapchain	An OS structure neccessary for online rendering. Provides VkImage and VkImageViews that can be drawn to and presented to the user. The swapchain is where values such as vsync are chosen.
VkCommandPool	Used to allocate VkCommandBuffers from GPU memory.
VkCommandBuffer	Used to issue GPU commands. Unlike OpenGL, we cannot issue commands directly for drawing. We add a number of comamnds to the buffer, such as binding descriptor sets and executing draw calls. You want at least 2 command buffers, as you cannot reset one that is currently being used by the GPU to render.
VkQueue	Location on the GPU where the buffered commands are executed. Generally there are 3 queues in use in rendering software, a draw queue, an asynchronous compute queue, and a data transfer queue.
VkFence	Used for GPU to CPU communication. The CPU must wait until the GPU has finished executing to submit more work, and the fence is set by the GPU at set times, allowing the CPU to wait for execution to finish.
VkSemaphore	Used for GPU to GPU synchronization. This can be used to define the order in which operations should happen on the GPU, rather than leaving it up to chance.

Table 4.1: The most common Vulkan terms, and the most important to understanding the renderer. Especially notable are the VkImage, VkDevice, and Swapchain.

which tell our pipelines what data will be available and how it will be used. We can also create our buffers for any scene data, like camera position, light positions, and so on. We also create all of our pipelines, which are the combination of vertex and fragment shaders. It should be mentioned that a pipeline is analogous to what we've called a material in Vulkan, since we can't swap out vertex and fragment shaders on the fly like OpenGL. Finally, we can load all of our models, materials, and textures that are in our scene.

4.2.2 Per Frame

A few things must be handled in each frame. First, we must poll all the events from SDL2, which tells us if any input has came from the user, if they want to close the window, if any resizing has happened, and anything else the user might have done.

After handling input we then update any objects that may have moved, the camera position, and any other parameters that changed between frames. We then want to wait for the rendering fence to complete, which lets us know the GPU has finished rendering the last frame. Once we've waited for the fence, we can clear all of the descriptors from the previous frame. We then need to ask the swapchain for the next image and proceed to reset the render fence and the current frame's command buffer.

Once we've reset the command buffer we can begin the current frame's commands. As the image we get from the swapchain is not in a valid format to write to, since we are using dynamic rendering, we first have to transition both the depth buffer and the color buffer from an undefined layout to an image we can write to. After the transition, we can bind each of our render pipelines, bind the descriptor sets, set any push constants, and then render all the objects for a pipeline. Because the overhead for switching pipelines is fairly high, we can sort all the objects in the scene by their material, which minimizes the number of pipeline switching. At this point, each object is handled by the shaders as described in the render pipeline.

One important thing to note is that some of the largest overhead while rendering comes when transfering the data from the CPU to the GPU. Additionally, the GPU has a limited amount VRAM that can be used, so multiple applications running at the same time have

to share the limited resources. If the available VRAM is not adequate, we cannot take advantage of cached textures and vertex buffers.

4.2.3 Cleanup

At the end of the program, everything needs to be cleaned up, and all of the handles should be cleaned up as well. The simplest way to handle this is with a deletion queue, and to just clean up in the reverse order that everything was created. We also need to wait for the current frame to finish, which we can do by waiting for the render fence just like if we had started a new frame.

CHAPTER 5
STREAMING

5.1 General Streaming

The renderer, at its core, it built to allow for remote user connections. These remote users each have their own view of the scene, requiring facilitation of connections between the host and remote. A general diagram of the relationship between client and host can be seen in figure 5.1. After the host starts up it needs to listen for incoming connection requests. The remote program runs a very simplified version of the renderer, bypassing the need for Vulkan to run. This is possible by utilizing SDL2's ability to display a texture to the window, without directly using a graphics library, although SDL2 uses OpenGL for its backend by default. Setting up Vulkan as a fully fledged renderer for the client is unnecessary, as we only need to display the image from the host.

When the remote user joins, the host creates a second camera in the world and begins rendering for both, rendering each user's view every frame. By handling the rendering in this manner, we can avoid the overhead of loading the scene that would happen if we had to run both users' visions of the area separately, especially if the scene almost fills the host's VRAM.

To facilitate the connection between the host and client, a lightweight protocol built on top of TCP is used. The first 16 bytes consist of two, 64-bit integers, which describe the data. The first integer tells the client how large the JPG file is which depends on the compression level and method. The second integer describes the row pitch for the incoming data, which is important for the client to correctly decode the JPG file upon arrival. The row pitch cannot be predetermined as the VkImage may have an arbitrary pitch, determined by the graphics card manufacturer.

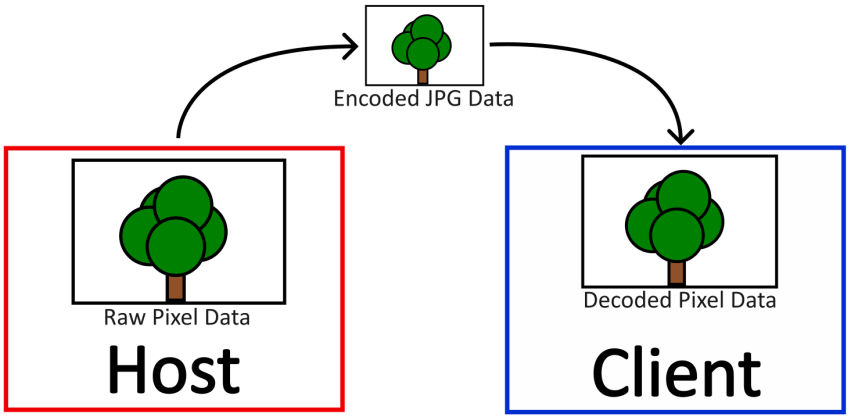

Fig. 5.1: The relationship between host and client is shown. The raw pixel data from the host's GPU is encoded to JPG format before transfer to the client, where it will be decoded back into the raw pixels.

5.2 Host

When the host begins rendering, the default behavior is to only render for themselves. The other user's camera does not exist yet in the renderer. When a user begins a connection, a new set of controls and a camera are created for that user, on the host's machine. The render loop then switches slightly once the client is connected, rendering for both cameras each frame, but keeping the output separate for each. As the frame is displayed to the host user, the VkImage 4.1 for the remote user is outputted to the raw data on the CPU, which is then converted into a compressed image format, then transferred to the remote user, while the next frame begins rendering. The communication is handled on a separate background thread to mitigate rendering time impact.

There is slight latency on the image between the render and transfer to the user, which can be seen in the results chapter. The latency while on the same network is minimal, however, as shown in the results.

5.3 Client

On the client's side, when the renderer starts up, it attempts communication with a preset host. The client uses a simplified version of the renderer, focused mainly on the

display of the transferred images. The client does not require the full Vulkan rendering process, instead being able to just display the fully rendered frame received from the host. It should be noted that the client is unable to interact with the scene parameters that the host can, due to the rendering only happening on the host side. One example of these scene parameters are the position and intensity of lights, and the selected compute shader generating the background.

The client first reads the header describing the file, then creates a buffer and copies the file into it. After the image fully transfers, the buffer is decoded out of its JPG format and into the raw pixels to be used by the client.

The client takes the raw pixels and utilizes a streamed SDL texture, which allows fast access between both the CPU and GPU. This is essential as we will be transferring every frame.

Due to network conditions, occasionally the client may read the header incorrectly by reading the next frame as part of the current. When this happens, the client detects that the transfer has encountered an error and resets the connection between itself and the host, to start with a clean slate. The impact of this reset is very small, only noticeable by logging it in the console window.

5.4 Render Loop Changes

A few changes are made to the render loop to handle the requirements added by rendering for a second user. One major change is that for the remote render loop, we use the OS swapchain to get a VkImage and VkImageView to render on, which then is displayed on the screen. For the client, we treat the loop more similar to an offline renderer, where we can create a VkImage to have our own 'virtual' swapchain instead. For this, we will instead have a second render call, where we will render the second camera to its own image, that is not displayed on the screen.

The client's render call is setup the same way as the main loop, other than the final copy from the render VkImage to the swapchain. However, this loop does need to wait for the fences and semaphores of the original render, as the GPU may still be busy rendering the

host's image. To allow for the sharing of memory between the host and client render loops, the same device is used for both. The remote render loop is initialized at the beginning of the program, alongside the regular loop, but can be left unused if the remote doesn't connect. It also can go back to an idle state if the remote user disconnects, so the host is not rendering useless frames.

CHAPTER 6

RESULTS

6.1 Introduction

To demonstrate the effectiveness of this method of rendering, metrics, and results for both the host and client are shown. Key metrics are the frame rate impact on the host, as well as the difference between the original frame rate of the client when it renders for itself, as opposed to the frame rate when utilizing the remote setup.

All testing was performed using a desktop computer with a Nvidia RTX 3080 Ti as the host, and a laptop with a Nvidia GTX 1050 as the client.

6.2 Host Performance

In this section, the performance on the host's computer is shown. The main effect on the host was found to be the frame rate.

6.2.1 Frame Rate Impact

The increased overhead that comes when rendering for the additional user depends on the complexity of the scene. On simple scenes, the impact is negligible, with more complex scenes having a larger impact. For a single model, the draw time for the host averaged around 0.05 ms. Adding a second render call to render for the client added an additional 0.05 ms, totaling an average of 0.1 ms. This makes the rendering overhead virtually unnoticeable, as for a 60 hz monitor, you have around 16 ms available to render. Table 6.1 breaks down the average frame rates, for a complex and simple scene.

For the complex scene, it can be seen in table 6.1 that adding the second render for the remote user has an obvious effect, and almost halves the frame rate for the host. However, as seen in the simple scene, the impact of the client is unnoticeable when the host is not under strain. As shown in the complex scene, when the host's GPU already cannot keep

Scenario	Frame Time (ms)	FPS
Host, Simple	6.2694	160
Host, Complex	26.4661	38
Both, Simple	6.2133	160
Both, Complex	38.0096	26

Table 6.1: The average frame timing for each of the host's scenarios is shown. Rendering for both the client and host does not have a noticeable effect on the host's frame rate, as long as the host still has leftover idle time between frames.

up with the scene rendering, adding another user will have a notable performance effect for the host, but for GPUs that are underutilized, such as in the simple scene, the impact of the remote user is completely unnoticeable. This is the strength of this rendering setup, as it is not built to share between equal or similar GPUs, but instead, with a powerful GPU that is underutilized, with another that struggles to render the scene on it's own.

The encoding of the image takes an average of 21 ms to perform. This encoding time is performed on a background thread to ensure the main rendering thread is unblocked, thereby removing the potential impact the encoding could pose for the host's frame rate.

6.3 Remote Performance

In this section, the performance of the remote user's computer was evaluated. The metrics used to measure performance were the frame rate differences between rendering locally vs. remote, latency between render time and present time, and visual examples of the effects of varying levels of compression on the images.

6.3.1 Frame Rate Impact

For the complex scene, the client computer has an original frame time of 137.5766 ms, or about 7 FPS. The heavy complexity of the scene renders the client unable to render for itself at an adequate speed.

After connecting to the host, the client has an 'effective' frame rate of 26 FPS. We use the term 'effective' frame rate, as the true frame rate of the client is 60 FPS, but that does not properly describe the speed. This is because the client will render the last frame twice

if the current frame is not ready. The 'effective' frame rate refers to the speed at which new images are displayed on the screen, rather than repeated images.

6.3.2 Latency

The latency between render time and presentation time for the client is shown below. The latency is measured as the time elapsed between completing the render of the frame and the frame's presentation on the client device. Ideally, this would remain close to the refresh rate of the monitor and can be thought of as the 'true' frame rate of the rendering process.

An important note is that the latency is not only dependent on the encoding, decoding, and presentation time, but the conditions of the network as well. The results shown here are tested on a local network, to minimize the impact of outside network conditions. The latency for both the host and client's internet connections would have the biggest impact on this metric, as the latency rising from the actual preparation on both sides is minimal. Because the tests performed were on a local network, the transfer time between host and client was an average of 28.8638 ms.

The other large effect on latency between client and host was from the encoding of the image, which took an average of 21.0000 ms. This led the latency between the client and the host to be 49.8638 ms. For the complex scene, this meant that the client was generally 2-3 frames behind the host. On the simple scene, any frames that come during the encoding and transfer to client are dropped, to ensure that the client does not get too far behind the host. On the complex scene, as the encoding was faster than the frame update time, the client does not have any dropped frames.

6.3.3 Compression Effects

The effects of the image encoding are shown in A.1. As the encoding used is JPG, the effects of the compression are especially noticeable in the gradient background. These figures show the visual effect on the final image, which the effect on the size is shown in table 6.2. For the local networks tested, the network was sufficient to handle a JPG compressed

Quality Level	Size (MB)	Req. Network Speed (60 FPS)
Raw Pixels	5.32	319.2 MB/s
90	0.0798	4.788 MB/s
75	0.0586	3.516 MB/s
50	0.0479	2.874 MB/s
25	0.0416	2.496 MB/s

Table 6.2: The effect of reducing the quality during JPG encoding on file sizes, as well as the raw pixel data before encoding. Note that the size reduction is not linear with the quality variable.

with a quality of 90%. This was chosen as the default as the compression artifacts were very obvious with higher compressed images, while only having diminishing returns on the actual file size.

6.4 Discussion

The renderer demonstrated provides the capability to share computational power with another user. The effect on the host is more noticeable in scenes with high complexity, but is unnoticeable in scenarios with relatively low complexity. One key factor that is not easily shown in the results here, is the ability of the host to utilize Vulkan functionality that may not be available to the client, rather than only speed increases. In a fully fledged renderer, the host can perform hardware raytracing and other optional Vulkan functionality that the client's hardware is incapable of performing. The data shown here speaks to the benefit of sharing the computational speed increases from more expensive hardware, but the ability of the renderer to share the built in hardware features, like raytracing, with another user are a major advatage to this method of distributed rendering. The architecture can remain the same, and improvements to the core rendering functionality can be added, while the client only has to run the low-impact client-side software.

All the code is available at https://github.com/tganderson0/crow-engine.

6.5 Future Work

Building on this system, a hybrid renderer could be created, one that utilizes the

GPU capacity of both client and host. A common technique used in real time renderers is to use partial ray-tracing, or using ray-tracing to help get the most important details accurate, while using regular rasterization for the rest of the scene. In this manner, the most complicated and intensive work could be done by the host, while the client can begin to render the simpler geometry or UI, which does not require powerful hardware.

Additionally, the core rendering capabilities will continue to be developed by implementing skinned meshes, animations, a physics system, and more to allow it to stand on more equal ground with modern rendering software, instead of existing as a prototype physically based renderer.

REFERENCES

[1] C. Tomasi, "A simple camera model," https://courses.cs.duke.edu//fall16/compsci527/notes/camera-model.pdf, accessed: 2024-02-08.

[2] A. Kugler, "The setup for triangle rasterization." in *Workshop on Graphics Hardware*, 1996, pp. 49–58.

[3] K. Akeley, "Reality engine graphics," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, 1993, pp. 109–116.

[4] "Khronos gltf 2.0 released as an iso/iec international standard," https://www.khronos.org/news/press/khronos-gltf-2.0-released-as-an-iso-iec-international-standard, accessed: 2024-02-08.

[5] B. T. Phong, "Illumination for computer generated pictures," *Commun. ACM*, vol. 18, no. 6, p. 311–317, jun 1975. [Online]. Available: https://doi.org/10.1145/360825.360839

[6] B. Burley and W. D. A. Studios, "Physically-based shading at disney," in *Acm Siggraph*, vol. 2012. vol. 2012, 2012, pp. 1–7.

[7] P. Beckmann and A. Spizzichino, "The scattering of electromagnetic waves from rough surfaces," *Norwood*, 1987.

[8] J. F. Blinn, "Models of light reflection for computer synthesized pictures," in *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, 1977, pp. 192–198.

[9] R. L. Cook and K. E. Torrance, "A reflectance model for computer graphics," *ACM Transactions on Graphics (ToG)*, vol. 1, no. 1, pp. 7–24, 1982.

[10] D. S. Immel, M. F. Cohen, and D. P. Greenberg, "A radiosity method for non-diffuse environments," *Acm Siggraph Computer Graphics*, vol. 20, no. 4, pp. 133–142, 1986.

[11] J. T. Kajiya, "The rendering equation," in *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, 1986, pp. 143–150.

[12] K. E. Torrance and E. M. Sparrow, "Theory for off-specular reflection from roughened surfaces," *Josa*, vol. 57, no. 9, pp. 1105–1114, 1967.

[13] B. Smith, "Geometrical shadowing of a random rough surface," *IEEE transactions on antennas and propagation*, vol. 15, no. 5, pp. 668–671, 1967.

[14] C. Schlick, "An inexpensive brdf model for physically-based rendering," in *Computer graphics forum*, vol. 13, no. 3. Wiley Online Library, 1994, pp. 233–246.

[15] B. Walter, S. R. Marschner, H. Li, and K. E. Torrance, "Microfacet models for refraction through rough surfaces," in *Proceedings of the 18th Eurographics conference on Rendering Techniques*, 2007, pp. 195–206.

[16] B. Karis and E. Games, "Real shading in unreal engine 4," *Proc. Physically Based Shading Theory Practice*, vol. 4, no. 3, p. 1, 2013.

APPENDICES

APPENDIX A

Rendered Outputs

A.1 Encoding Quality Effects

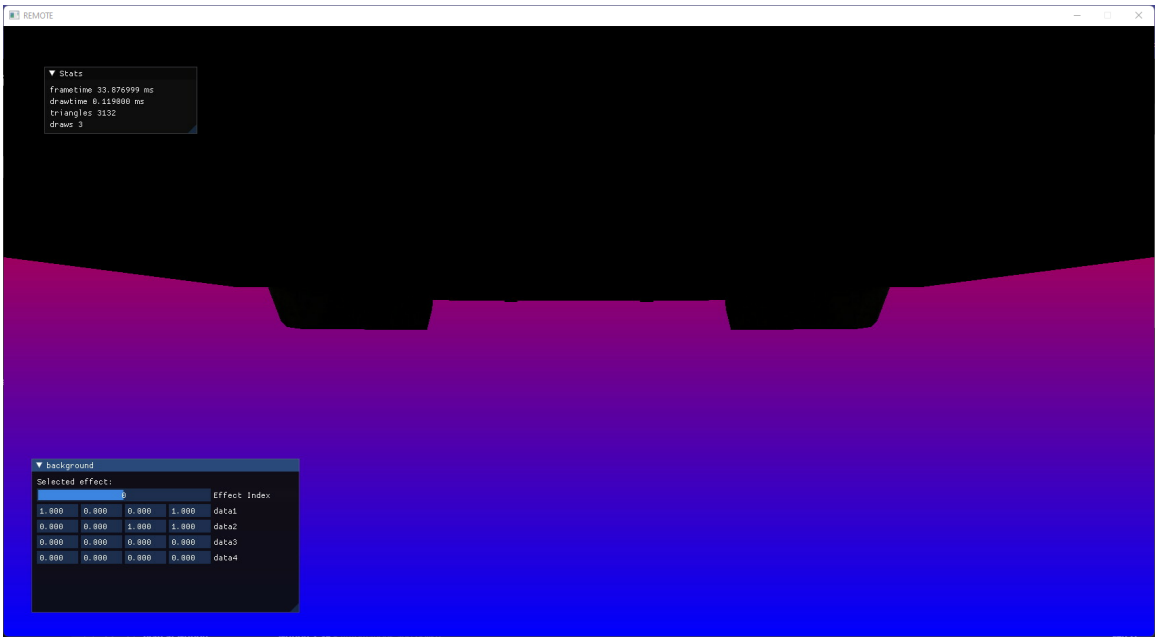

Fig. A.1: Example render with a quality of 90. With this quality setting for the JPG encoding, visual artifacts are generally unseen, although perceptible in the gradient.

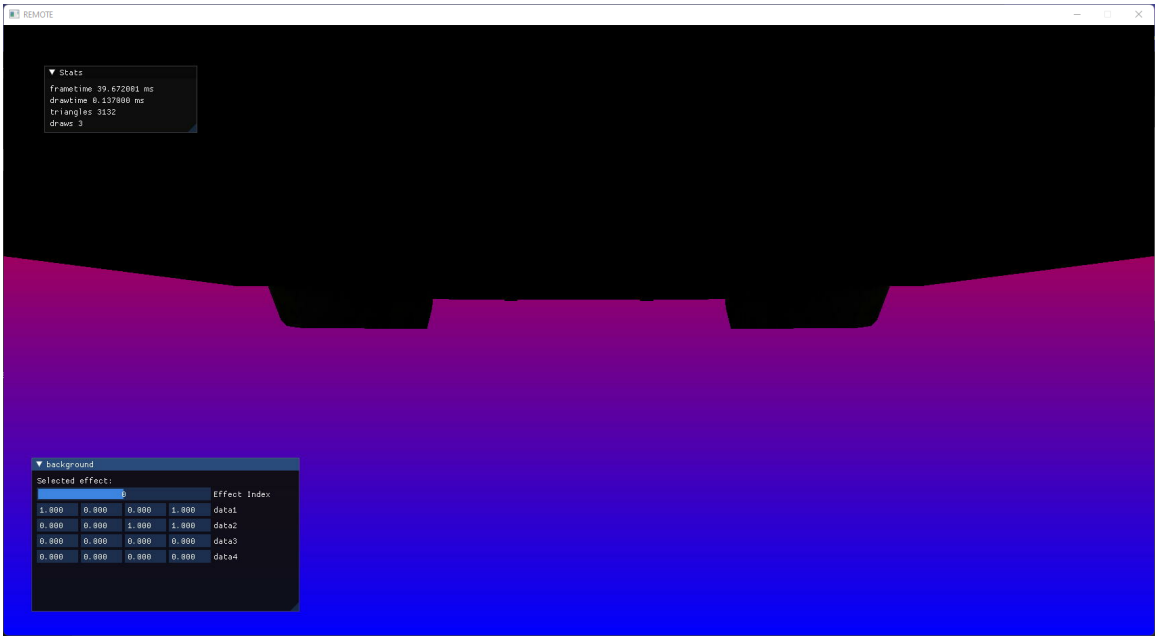

Fig. A.2: Example render with a quality of 75. Decreasing the quality to this level starts to create more apparent visual artifacts in the transmitted image.

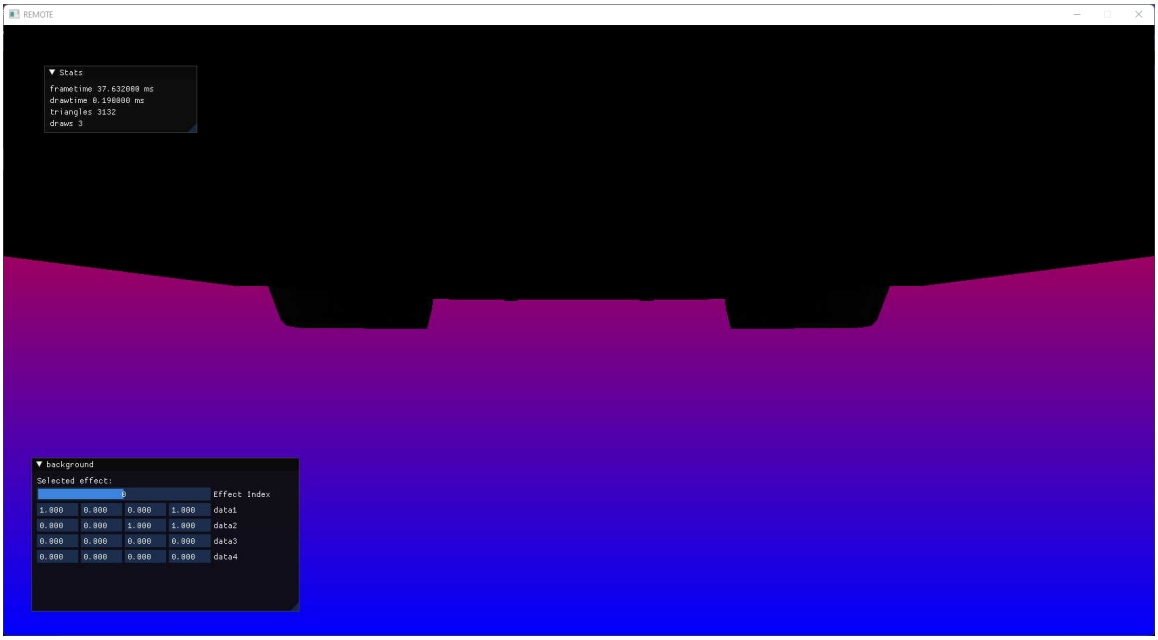

Fig. A.3: Example render with a quality of 50. The visual artifacts are even more visible, especially in the gradient.

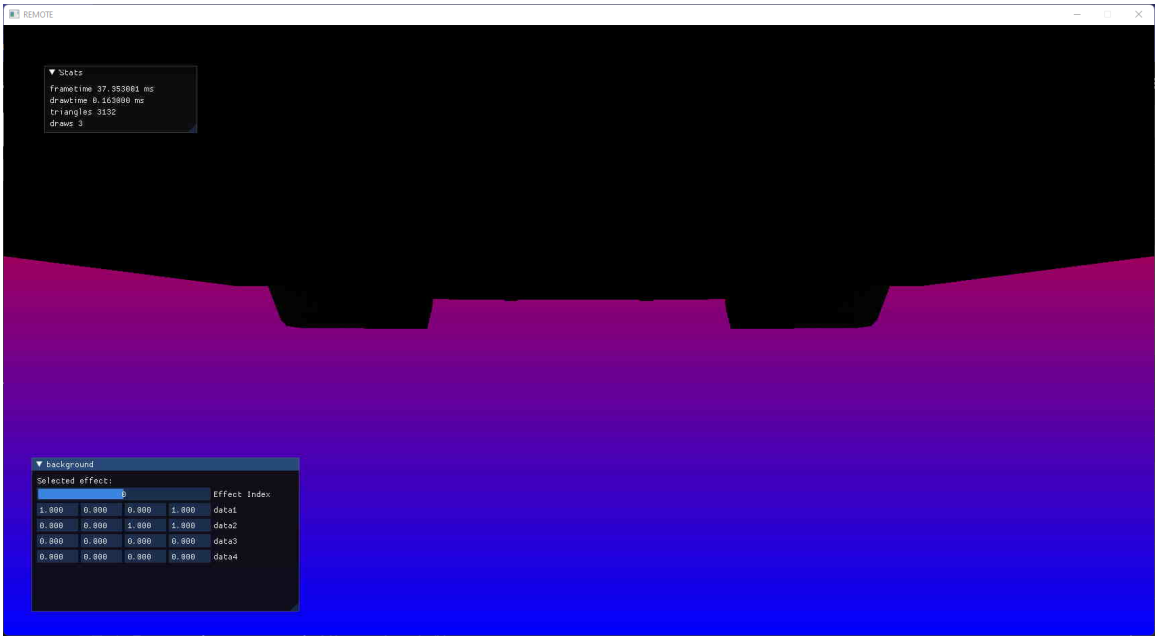

Fig. A.4: Example render with a quality of 25. At this quality, the visual artifacts are obvious, and the solid black area has clear errors near the border of the dark and lighter areas.

CURRICULUM VITAE

Taylor Gordon Anderson

Published Conference Papers

- Stealth Centric A*: Bio-Inspired Navigation for Ground Robots, Ryan Anderson, Taylor Anderson, Carter Bailey, Jeffrey Anderson, and Mario Harper, in *Proc. IEEE Int. Conf. on Robotic Computing (IRC)*, 2023.

Published Journal Articles

- Charger Reservation Web Application, Ryan Anderson, Jeffrey Anderson, Taylor Anderson, and Mario Harper, *Software Impacts*, vol. 18, pp. 100589, 2023
- Stealth Centric Autonomous Robot Simulator (SCARS), Jeffrey Anderson, Ryan Anderson, Taylor Anderson, Carter Bailey, and Mario Harper, *Software Impacts*, vol. 16, pp. 100497, 2023
- Power and transportation collection and visualization, Ryan Anderson, Taylor Anderson, and Mario Harper, *Software Impacts*, vol. 14, pp. 100386, 2022