BSc (Hons) Computer Games Technology

Real-Time Physically-Based Rendering

Stuart D. Adams
B00265262

22/03/2019

Supervisor: Marco Gilardi

Declaration

This dissertation is submitted in partial fulfillment of the requirements for the degree of BSc (Hons) Computer Games Technology in the University of the West of Scotland.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself. Following normal academic conventions, I have made due acknowledgement to the work of others.

Name:
STUART ADAMS

Signature:

Date:
28/03/2019

To be completed in full

Surname **Adams**
First Name **Initials** **Stuart** **S. D. A.**
Borrower ID Number **B00265262**
Course Code **COMPSCI**
Course Description **BSc (Hons) Computer Games Technology**
Project Supervisor **Marco Gilardi**
Dissertation Title **Real-Time Physically-Based Rendering**
Session **2018-19**

COMPUTING HONOURS PROJECT SPECIFICATION FORM

Project Title: Real-Time Physically-Based Rendering

Student: Stuart Adams

Banner ID: B00265262

Supervisor: Marco Gilardi

Moderator: Paul Keir

Outline of Project:

This project will demonstrate physically-based rendering techniques for replicating the properties of light and material interaction. A new renderer will be developed that employs these techniques. This renderer will ensure that materials behave accurately in any lighting environment. When the renderer is complete, the effectiveness of the implementation will be measured through surveys that will ask participants to compare renders from the new renderer against commercially available game engines.

A Passable Project will:

Demonstrate a microfacet-based lighting model. Allow per-fragment control over material properties such as albedo, metalness, roughness and normals. HDR with Gamma correction must be implemented. A microfacet specular BRDF must be implemented, respecting energy conservation and allowing fresnel reflections.

A First Class Project will:

Epic Games' split-sum approximation technique must be used to achieve Specular Image-Based Lighting. A reusable, well-documented C++ framework must be developed, abstracting the OpenGL implementation details.

Marking Scheme	**Marks**
Introduction	10
Literature Review	15
Software Design	20
Implementation	25
Evaluation	15
Conclusion	10
Critical Self-Appraisal	5

Mathematical Notation

v	View vector
l	Incident light vector
n	Surface normal
h	Halfway vector
p	Intersection of **v** with the closest object surface
L	Radiace
L_i	Incoming radiance
L_o	Outgoing radiance
f	BRDF
f_d	Diffuse component of a BRDF
f_r	Specular component of a BRDF
ρ	Albedo of a surface
α	Material roughness
$\chi^+(a)$	Heaviside function: 1 if a > 0 and 0 if a <= 0

Acknowledgements

I would like to thank my supervisor Dr. Marco Gilardi for his guidance and patience. His experience with mathematics and computer graphics helped my project realise its full potential, and his willingness to explain unfamiliar concepts helped me secure a stronger understanding of the underlying theory. I was able to approach this project with an academic focus thanks to his advice and recommendations. What I have learned from him will help me in future research-based work in my professional career.

My moderator Dr. Paul Keir has always encouraged me to reach higher and to tackle interesting technical problems. His modules this year exposed me to many interesting graphics APIs, inspiring the design of this project. He was always happy to answer any of my many tedious questions about graphics hardware, cross-platform software and modern C++.

Finally, I would like to express my gratitude to UWS alumni James Moore and Christoffer Pettersson. Without James' continued mentorship and support I would still struggle to engage with C++ programming. Christoffer offered me frequent graphics programming advice and introduced me to graphics debugging tools that accelerated this project's development.

Summary

This dissertation discusses techniques for achieving photo-realism in real-time applications. The foundations of light-matter interaction are reviewed before discussing how these observations can be applied in the shading process. The design and implementation of a real-time renderer is presented, exemplifying the techniques discussed. The implementation is evaluated by video game players for the realism of its shading using a questionnaire.

Respondents to the questionnaire were asked to rate the realism of the project's renderings, alongside similar renderings produced by UE4. Their responses were analysed to evaluate the effectiveness of the project's implementation. The analysis showed that the implementation was successful in realistic shading that the target population considered comparable in quality to UE4.

The project is a rendering engine written in C++17 and OpenGL. It implements advanced rendering techniques used by commercial game engines, including a physically-based BRDF and image-based lighting with diffuse / specular light probes. Supplemental GLSL code is included.

Contents

1. Introduction

Consumers are demanding higher graphical fidelity. The highest factor influencing a consumer's decision to purchase a game is the quality of its graphics (Entertainment Software Association, 2017). Modern games use physically-based rendering to meet this demand. Most modern game engines, including Unreal (Karis, 2013), Unity (Pranckevičius, 2014a), and Frostbite (Lagarde and Rousiers, 2014), use physically-based rendering techniques in their default shading models.

Physically-based rendering (PBR) is the practice of simulating the physics of light and matter interaction to achieve realistic shading. When working with empirical lighting models, such as the *Phong reflectance mode*l (Phong, 1975), artists must reconfigure materials to produce high-quality images in different lighting conditions. PBR techniques can create physically-plausible images regardless of the materials or lighting environment (Pranckevičius, 2014a). These techniques are now commonplace in real-time and offline rendering (Pharr et al., 2017).

The aim of this research is to evaluate physically-based rendering techniques and use them to develop a real-time renderer. The new renderer demonstrates PBR and allows the user to configure the lighting environment. A survey was taken to determine the effectiveness of the implementation, asking participants to judge 10 rendered images for their photo-realism. Half of the images were from a commercial game engine. The participants did not know which renderer produced each image. This allowed the new renderer to be evaluated against a state-of-the-art commercial game engine.

The evaluation stage of this research had several ethical considerations. Feedback from the survey was recorded anonymously. All subjects signed a consent form, agreeing to be observed and have data collected based on these observations. Subjects could interrupt the session. Care was taken to avoid exposing the subject disturbing imagery. Fabricating or falsifying data in the research would have been a major ethical violation. Attention was paid to honour all intellectual property used by giving appropriate credits.

1.1. Contributions

This project contributes to the field of computer graphics by showing a simple, practical implementation of physically-based rendering in OpenGL. It tests the implementation against a leading commercial game engine to make clear the benefits of a physically-based approach, and to determine if the development efforts were successful.

1.2. Source Code

The source code for this project can be accessed at the following GitHub repository:

https://github.com/stuartdadams/moka

The source code comprises original programming and sample code provided by research papers. Proper code attribution is given in the source files. The project uses a CMake build system. CMake is a C++ Makefiles/Solution generator. It is available on Windows and most Linux systems as a package. Listing 1.1 demonstrates how to clone the repository and generate solution files. The Hunter package manager (Baratov, 2019) will download and install all dependencies required by Moka, so the development environment will either need an existing Hunter cache or an internet connection to compile Moka.

```
git clone https://github.com/StuartDAdams/moka.git
cd moka
mkdir build
cd build
cmake ../
```

Listing 1.1: Cloning the moka repo and generating solution files.

The program can be configured using the examples/assets/config.json file. The configuration file defines paths to the HDRI environment map and glTF asset that the application will import and render.

2. Theoretical Background

Empirical lighting models are cheap and produce appealing results, but achieving greater realism requires creating a correct distribution of reflected light. Modern game engines use a better model of a material's microscopic structure and apply electromagnetic theory (Lengyel, 2014). To understand the interaction of light and matter requires a basic understanding of light.

2.1. Radiometry

Radiometry is the field concerned with studying the physical transmission of light. It deals with the measurement of electromagnetic radiation. Light is modeled as an electromagnetic transverse wave - a wave that oscillates the magnetic and electric fields perpendicular to the direction of its propagation (Hoffman, 2013). The oscillations of both fields are coupled, as demonstrated in Figure 2.1.

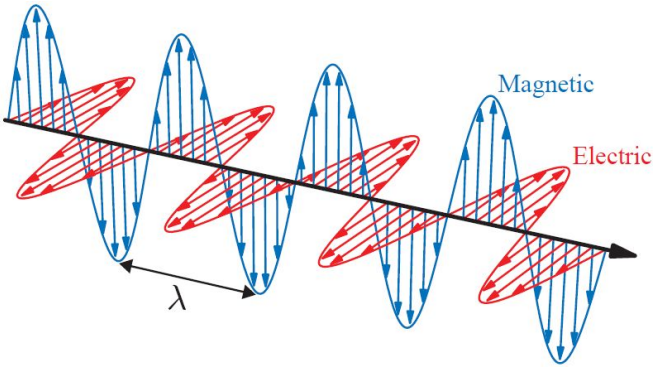


Figure 2.1. Light, an electromagnetic transverse wave (Akenine-Möller et al., 2018).

Figure 2.1 depicts the simplest wave possible, a sine function. It has one *wavelength*, denoted with λ. Wavelength is the distance over which the wave's shape repeats. Waves with different wavelengths have different properties. The perceived colour of light is related to its wavelength. Light with a single wavelength is *monochromatic*. Most light waves are *polychromatic*, containing many different wavelengths. Human vision can only perceive a subset of this range, from 400 nanometers for violet light, to 700 nanometers for red light. This range, visible in Figure 2.2, is of interest for shading (Hoffman, 2013).

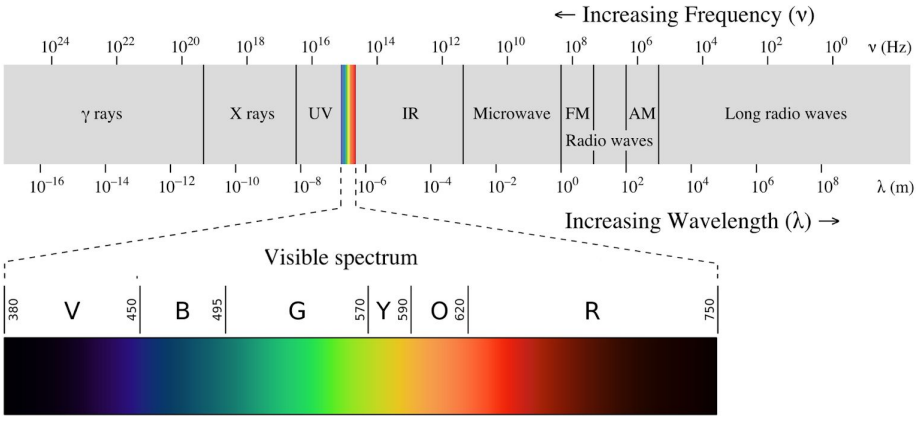


Figure 2.2. The visible spectrum (Akenine-Möller et al., 2018)

2.1.1. Basic Quantities

There are four radiometric quantities of interest in rendering: *flux*, *irradiance / radiant exitance*, *intensity*, and *radiance* (Pharr et al., 2017). Each radiometric quantity can be derived from energy by taking successive limits over time and direction.

2.1.1.1. Energy

Energy is measured in *joules, J*. Sources of illumination emit photons, each of which is at a specific wavelength and carries a specific amount of energy. The following quantities can be thought of as different ways of measuring photons. The amount of energy carried by a photon can be calculated:

$$Q = \frac{hc}{\lambda}. \tag{2.1}$$

Where Q is the energy carried by a photon, h is Planck's constant, $h \approx 6.626 \times 10^{-34}\ m^2\ kg/s$, and c is the speed of light, $299,472,458\ m/s$.

2.1.1.2. Flux

Graphics programmers are most interested in measuring light at an instant (Pharr et al., 2017). *Radiant flux*, Φ, also known as *power*, is the amount of energy passing through a surface per unit time. Radiant flux can be calculated by taking the limit of differential energy per differential time:

$$\Phi = \lim_{\Delta A \to 0} \frac{\Delta Q}{\Delta t} = \frac{\mathrm{d}Q}{\mathrm{d}t}. \tag{2.2}$$

Its units are *J/s*, or *watts, w*. Given flux as a function of time, it is possible to integrate over a range of times to compute the total energy:

$$Q = \int_{t_0}^{t_1} \Phi(t)\mathrm{d}t. \tag{2.3}$$

Flux describes the total emission of a light source (Pharr et al., 2017). When Figure 2.3 is considered, the amount of energy passing through any point on the larger sphere is less than the amount of energy passing through the smaller sphere, although the total flux measured by either two spheres is the same.

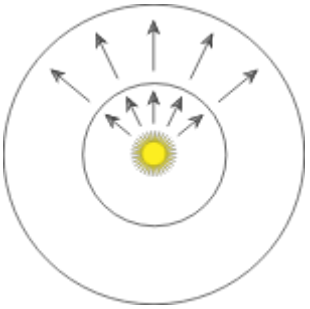


Figure 2.3. Radiant flux is measured using spherical surfaces that surround a point light (Pharr et al., 2017)

2.1.1.3. Irradiance and Radiant Exitance

Measuring flux requires an area over which photons per time can be observed. The average density of radiant flux over the finite area *A* can be calculated by

$$E = \Phi / A. \tag{2.4}$$

This quantity is *irradiance, E*, the area density of flux arriving at a surface, or *radiant exitance, M*, the area density of flux leaving a surface. The term irradiance can refer to both cases. These measurements have units of W/m^2.

If Figure 2.3 is considered, the irradiance at any point on the outer sphere will be less than the irradiance on any point on the inner sphere, due to the increase in surface area. If a point source is illuminating in all directions, then for a sphere with radius *r*,

$$E = \frac{\Phi}{4\pi r^2}. \tag{2.5}$$

Equation 2.5 explains why the energy received from a point light falls off with the squared distance from the light. Irradiance and radiant exitance can be defined by taking the limit of differential power per differential area at point p:

$$E(\mathrm{p}) = \lim_{\Delta A \to 0} \frac{\Delta \Phi(\mathrm{p})}{\Delta A} = \frac{\mathrm{d}\Phi(\mathrm{p})}{\mathrm{d}A}. \tag{2.6}$$

Irradiance can be integrated over an area to find radiant flux:

$$\Phi = \int_A E(\mathrm{p}) \mathrm{d}A. \tag{2.7}$$

2.1.1.4. Solid Angle and Intensity

Radiant intensity and radiance are defined in terms of *steradians, sr* (Lengyel, 2012). Steradians are used to measure *solid angles* - a three-dimensional extension of the concept of a planar angle. Steradians define the size of a set of continuous directions in three-dimensional space, similar to how radians define the size of a set of directions on a plane. Steradians are defined by the area of the intersection pitch on an enclosing sphere with radius 1. An angle of 2π radians covers a whole unit circle. A solid angle of 4π steradians covers a whole unit sphere (see Figure 2.4).

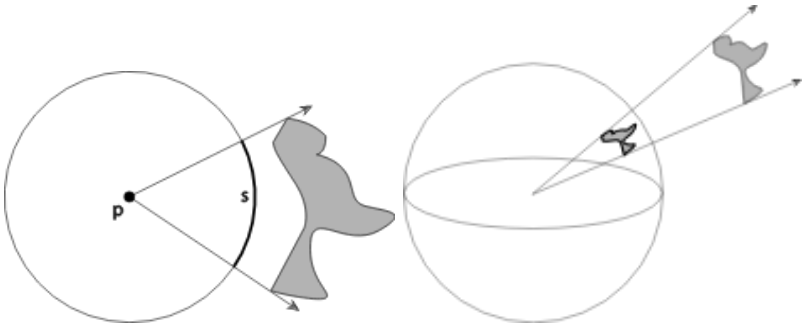


Figure 2.4. Planar angle and solid angle (Pharr et al., 2017)

Points on the unit sphere that encloses a central point *p* can describe all directions surrounding *p*. The symbol ω represents a normalized vector that will show the directions within the unit sphere. If Figure 2.3 is considered again, the angular density of emitted power, or intensity, can be calculated. It is denoted by *I*, and represents *W/sr*. Over the entire sphere:

$$I = \frac{\Phi}{4\pi r^2}. \tag{2.8}$$

To take the limit of a differential cone of directions:

$$I = \lim_{\Delta \omega \to 0} \frac{\Delta \Phi}{\Delta \omega} = \frac{\mathrm{d}\Phi}{\mathrm{d}\omega}. \tag{2.9}$$

Radiant flux can be recovered by integrating over a finite set of directions, Ω:

$$\Phi = \int_{\Omega} I(\omega)\mathrm{d}\omega. \tag{2.10}$$

2.1.1.5. Radiance

Radiance is the most important radiometric quantity. Radiance measures irradiance, or radiant exitance, with respect to solid angles (Lengyel, 2012). Radiance is defined by:

$$L(\mathrm{p},\omega) = \lim_{\Delta\omega\to 0} \frac{\Delta E_{\omega}(\mathrm{p})}{\Delta\omega} = \frac{\mathrm{d}E_{\omega}(\mathrm{p})}{\mathrm{d}\omega}. \tag{2.11}$$

Where E_{ω} denotes irradiance at the surface perpendicular to the direction ω. Radiance is the flux density per unit area, per unit solid angle. To define radiance in terms of flux:

$$L = \frac{\mathrm{d}\Phi}{\mathrm{d}\omega \mathrm{d}A^{\perp}}. \tag{2.12}$$

Where $\mathrm{d}A^{\perp}$ is the projected area of $\mathrm{d}A$ onto a hypothetical surface perpendicular to ω.

Radiance is the most fundamental of all radiometric quantities (Pharr et al., 2017). With radiance, all of the other quantities can be calculated in terms of integrals of radiance over areas and directions. In computer graphics equations, radiance often appears in the form $L_o(\mathrm{x},\mathrm{d})$ or $L_i(\mathrm{x},\mathrm{d})$, representing the radiance going out from **x** or entering it (Akenine-Möller et al., 2018). The direction vector, denoted by **d**, shows the ray's direction, which faces away from **x** by convention.

2.2. Surface Reflection

2.2.1. The BRDF

The traditional Phong reflectance model comprises three components: ambient, diffuse and specular light (Phong, 1975). Ambient light provides a background level of illumination and is reflected in all directions by everything in the scene. The diffuse and specular components are directional, reflecting light relative to the position and illumination of a particular light source. Diffuse light is modeled as Lambertian reflection from a surface scattered in all directions. Specular light is modeled as mirror-like highlights concentrated on the mirror direction. Few materials in the real world are perfectly Lambertian or mirror-like (Lengyel, 2012).

The *bidirectional reflectance distribution function* (BRDF) is a function that defines how light is reflected at a point on a surface (Hoffman, 2013). It is denoted as $f(\mathbf{l}, \mathbf{v})$. The purpose of a BRDF is to model how the radiant energy in a beam of light is redistributed when it strikes a surface (Lengyel, 2012). Some energy is absorbed by the surface; some may be transmitted through the surface; and the remaining energy is reflected. Reflected light is usually scattered in every direction in a non-uniform distribution. PBR renderers use BRDFs that produce a correct distribution of reflected light.

A BRDF is a 4D function over pairs of directions, **l** and **v**. Consider Figure 2.5: here, the BRDF describes how much light along **l** is scattered from the surface in the direction **v** (Pharr et al., 2017).

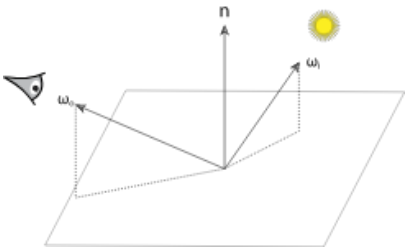


Figure 2.5. A BRDF where ω_i is **l**, and ω_o is **v** (Pharr et al., 2017)

Physically-based BRDFs must possess two qualities:

1. Reciprocity: $f(\mathbf{l}, \mathbf{v}) = f(\mathbf{v}, \mathbf{l})$
2. Energy conservation: The energy reflected by a surface is less than or equal to the energy it receives.

Reciprocity is required by offline rendering algorithms such as bidirectional path tracing. Real-time BRDFs often violate reciprocity without noticeable artifacts (Akenine-Möller et al., 2018). In real-time rendering, exact energy conservation is unnecessary, but an approximation of energy conservation is important to achieving realistic lighting (Akenine-Möller et al., 2018).

BRDFs were once defined for uniform surfaces, where the BRDF is assumed to be the same across the entire surface. A function that captures BRDF variations across a surface because of blemishes or surface details that change the reflective properties is called a *spatially varying BRDF* (SVBRDF) or *spatial BRDF* (SBRDF). This is typically implemented with a BRDF and texture mapping (McAllister, 2004). This case is exceedingly common however, and most publications use the term BRDF to assume a dependence on surface location (Akenine-Möller et al., 2018). To compute $L_o(\mathbf{p}, \mathbf{v})$, the BRDF needs to be incorporated in the *reflectance equation:*

$$L_o(\mathbf{p}, \mathbf{v}) = \int_{\mathbf{l} \in \Omega} f(\mathbf{l}, \mathbf{v}) L_i(\mathbf{p}, \mathbf{l})(\mathbf{n} \cdot \mathbf{l}) d\mathbf{l}. \tag{2.13}$$

The $\mathbf{l} \in \Omega$ subscript on the integral denotes that the function integrates over **l** vectors that lie within the unit hemisphere above the surface, which is centered on the surface normal **n**. The light direction **l** is swept over the hemisphere of incoming directions. Any incoming light direction can and will impart radiance on the surface. $d\mathbf{l}$ denotes the differential solid angle around **l** (Akenine-Möller et al., 2018).

The $L_i(\mathbf{p}, \mathbf{l})$ term in Equation 2.13 represents the incoming radiance from other parts of the scene. *Global illumination* algorithms calculate this value by simulating the propagation and reflection of light throughout the entire scene. This research focuses on *local illumination*, which uses the reflectance equation to calculate shading at each point on a surface. In local illumination algorithms, $L_i(\mathbf{p}, \mathbf{l})$ is given and does not have to be calculated. $L_i(\mathbf{p}, \mathbf{l})$ is often simplified to $L_i(\mathbf{l})$ for brevity (Akenine-Möller et al., 2018).

2.2.2. Fresnel Reflectance

The interaction of light with a surface follows the Fresnel equations developed by Augustin-Jean Fresnel (1788 - 1827). Light incident on a flat surface splits into a reflected part and a refracted part (Akenine-Möller et al., 2018). The Fresnel equations can be interpreted as a function $F(\theta_i)$, where θ_i is the angle between **n** and **l**. The notation $F(\mathbf{n}, \mathbf{l})$ is also used. This function has the following characteristics:

1. *Normal incidence*: when $\theta_i = 0°$, with the light perpendicular to the surface, $F(\theta_i)$, has a value, F_0, that is a property of the surface; the specular colour of the substance.

2. *Fresnel effect:* as θ_i increases, any light that hits the surface at glancing angles will cause the value of $F(\theta_i)$ to increase, reaching a value of 1 for all frequencies (white) when $\theta_i = 90°$ (see Figure 2.6).

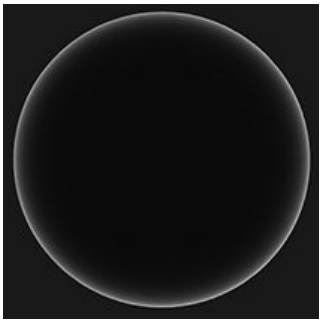


Figure 2.6. Fresnel effect (DeVries, 2016c)

Physically-based BRDFs define a function for calculating fresnel reflectance, denoted as *F*. Implementations of this function are explored in Section 3.2.3.

2.2.3. Microgeometry

A physically-based BRDF must compensate for microscopic imperfections. These irregularities are so small that a renderer cannot distinguish between them on a per-pixel basis. It is the aggregate behavior of the microfacets that determines the observed scattering (Pharr et al., 2017), so most BRDFs rely on a statistical distribution of normals to determine how many of *microfacets* are aligned with a certain direction. A microfacet is a tiny facet with a single microfacet normal. This distribution is defined by the *normal distribution function* (NDF), denoted as *D*. Their alignment relative to the surface of the object defines the roughness of that surface. Having misaligned microfacets results in a rough surface with a widespread, scattered specular reflection (see Figure 2.7). Smooth surfaces are more likely to reflect the incoming light in the same direction, resulting in sharper reflections. This is the primary effect of microgeometry.

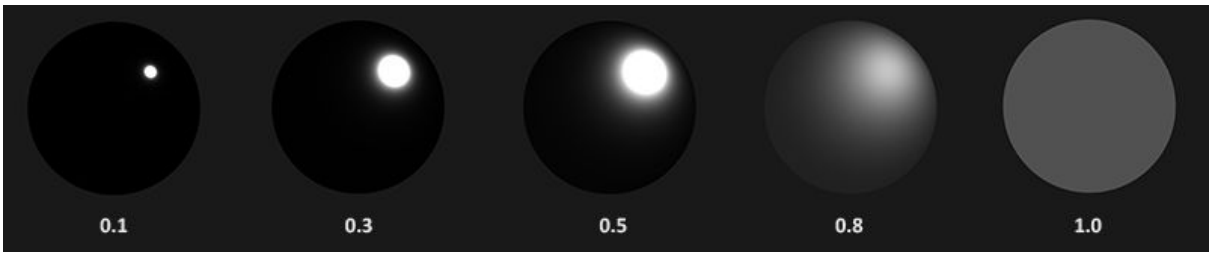


Figure 2.7. Visual result of increasing roughness with a normal distribution function (DeVries, 2016c)

Other effects include *shadowing*, where microscale surface details occlude the light source, and *masking*, where facets occlude each other from the camera (see Figure 2.8). These effects are approximated using a *geometry function,* denoted as *G*. Similar to the NDF, the geometry function takes a roughness parameter; but uses it to calculate the amount of shadowing and masking. Rough surfaces have a higher probability of both. The details of implementing a specular BRDF that exploits microfacet theory is explored in Section 3.2.

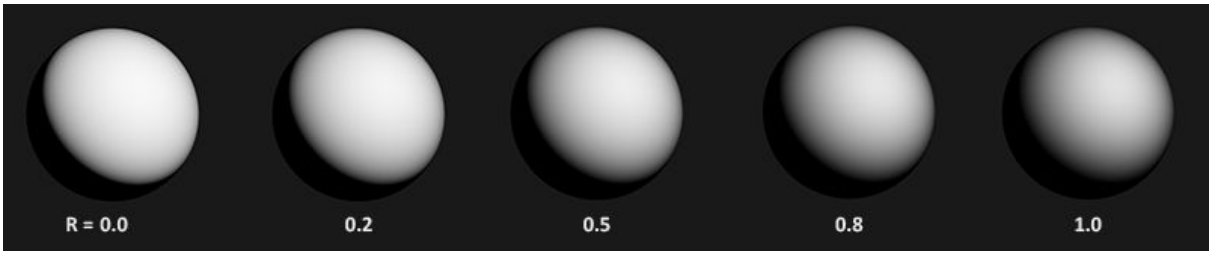


Figure 2.8. Visual result of increasing roughness with a geometry function (DeVries, 2016c)

3. Literature Review

The full BRDF can be split into two sections, each responsible for calculating a different part of the outgoing light from a surface. Consider this refactoring of Equation 2.13:

$$\int_{\mathbf{l}\in\Omega} f(\mathbf{l},\mathbf{v})L_i(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l} = \int_{\mathbf{l}\in\Omega} f_d(\mathbf{l},\mathbf{v})L_i(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l} + \int_{\mathbf{l}\in\Omega} f_r(\mathbf{l},\mathbf{v})L_i(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l} \tag{3.1}$$

Here, $f(\mathbf{l},\mathbf{v})$ defines the full BRDF, and $f_d(\mathbf{l},\mathbf{v})$ defines the diffuse BRDF; which is responsible for calculating the distribution of scattered light. Diffuse BRDFs are explored in Section 3.1. $f_r(\mathbf{l},\mathbf{v})$ defines the specular BRDF, responsible for calculating the distribution of reflected light. Specular BRDFs are explored in Section 3.2. GLSL shader code for each BRDF is also provided. Section 3.3 explores Image Based Lighting (IBL), a family of techniques where the lighting environment at a point is captured in all directions and evaluated with general BRDFs (Akenine-Möller et al., 2018).

3.1. Diffuse BRDF

3.1.1. Lambertian

In most real-time rendering applications, the diffuse component of light is calculated as though the surface was a perfect Lambertian reflector (Akenine-Möller et al., 2018). The Lambertian BRDF can be defined:

$$f_d(\mathbf{l},\mathbf{v}) = \frac{\rho}{\pi}. \tag{3.2}$$

The well-understood $\mathbf{n}\cdot\mathbf{l}$ factor that controls how much a surface is affected by incoming light is not part of the Lambertian BRDF, but it is part of Equation 3.1. The Lambert BRDF is notable for having a constant value; it assumes that scattered light loses all directionality. Lambertian diffuse is not always accurate. A surface that obeys Lambert's Law appears bright from all viewing directions. This is an inadequate approximation for several real-world materials with rough surfaces (Oren and Nayar, 1994). Many physically-based diffuse BRDFs have been proposed to model the diffuse component of more accurately than a typical Lambertian approach. Despite this, the Lambertian BRDF is still widely used in commercial video games. When used with a physically-based specular BRDF, the visual difference between the Lambertian BRDF and more accurate alternatives is subtle (see Figure 3.1).

After considering alternative diffuse BRDFs, Epic Games continued using Lambertian diffuse BRDF (Karis, 2013). Listing 3.1 shows the Lambertian BRDF defined in GLSL.

```
vec3 lambert(vec3 diffuse_color) {
    return diffuse_color / PI;
}
```

Listing 3.1: The Lambertian diffuse BRDF implemented with GLSL.

3.1.2. Oren-Nayar

Oren and Nayar proposed a modification of Lambertian reflectance that uses microfacet theory. They observed that Lambertian diffuse is not a perfect approximation for many materials, and proposed the modelling of surfaces as a collection of Lambertian microfacets (Oren and Nayar, 1994). Their model combines Lambertian diffuse with the Torrance-Sparrow model, which assumes the surface is composed of long, symmetric V-cavities. Each V-cavity has two opposing planar facets (Torrance and Sparrow, 1967). The model accounts for geometric phenomena that Lambert ignores, such as shadowing, masking and interreflections between facets (Oren and Nayar, 1994). Oren/Nayar can be defined:

$$f_d(\mathbf{l}, \mathbf{v}) = \frac{\rho}{\pi}(A + (B \cdot \max(0, \cos(\mathbf{n} \cdot \mathbf{l} - \mathbf{n} \cdot \mathbf{v})) \cdot \sin\beta_1 \cdot \tan\beta_2)). \tag{3.3}$$

Where:

$$\begin{aligned} A &= 1 - 0.5\frac{r^2}{r^2 + 0.33}, \\ B &= 1 - 0.45\frac{r^2}{r^2 + 0.09}, \\ \beta_1 &= \max(\mathbf{n} \cdot \mathbf{v}, \mathbf{n} \cdot \mathbf{l}), \\ \beta_2 &= \min(\mathbf{n} \cdot \mathbf{l}, \mathbf{n} \cdot \mathbf{v}). \end{aligned} \tag{3.4}$$

When $\mathbf{r} = 0$, $A = 1$, and $B = 0$, the Oren-Nayar BRDF can be simplified to the Lambertian model defined in Equation 3.2. Listing 3.2 shows a typical implementation of Oren/Nayar BRDF using GLSL.

```glsl
vec3 oren_nayar(
    vec3 diffuse_color,
    vec3 light_direction,
    vec3 view_direction,
    vec3 surface_normal,
    float roughness) {

    float n_dot_l = dot(surface_normal, light_direction);
    float n_dot_v = dot(surface_normal, view_direction);

    float r2 = roughness * roughness;

    float A = 1.0 - 0.5 * (r2 / (r2 + 0.33));
    float B = 0.45 * r2 / (r2 + 0.09);

    float beta_1 = max(n_dot_v, n_dot_l);
    float beta_2 = min(n_dot_l, n_dot_v);

    float intensity = A + (B * max(0, cos(n_dot_v - n_dot_l)) * sin(beta_1) * tan(beta_2);

    return lambert(diffuse_color) * intensity;
}
```

Listing 3.2: The Oren-Nayar diffuse BRDF implemented with GLSL.

3.1.3. Disney (Burley)

Disney considered using both Lambertian and Oren-Nayar diffuse BRDFs, but found that neither were able to meet their measured data (Burley, 2012). They developed a new diffuse BRDF that transitions between Fresnel shadow for smooth surfaces and adds a highlight for rough surfaces. The Disney BRDF can be defined:

$$f_d(\mathbf{l}, \mathbf{v}) = \frac{\rho}{\pi}(1 + F_{D90}(1 - (\mathbf{n} \cdot \mathbf{l}))^5)(1 + F_{D90}(1 - (\mathbf{n} \cdot \mathbf{v}))^5) \tag{3.5}$$

Where:

$$f_{\mathrm{D90}} = 0.5 + (\mathbf{l} \cdot \mathbf{h})^2 \alpha \tag{3.6}$$

Disney found this BRDF to be aesthetically pleasing and a reasonable match for their measured data, but noted that it is still empirical. An important caveat of this diffuse model is that it lacks energy conservation. The Frostbite engine introduces a modification that preserves its retro-reflective qualities while compensating for the gain of energy (Lagarde and Rousiers, 2014). Listing 3.3 shows the Disney function with Frostbite's renormalization factors. Figure 3.1 compares the Lambert BRDF with Disney's model.

```glsl
vec3 disney(vec3 diffuse_color,
    vec3 light_direction,
    vec3 view_direction,
    vec3 surface_normal,
    float roughness) {

    // https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf

    vec3 halfway = normalize(light_direction + view_direction);
    float n_dot_l = dot(surface_normal, light_direction);
    float n_dot_v = dot(surface_normal, view_direction);
    float l_dot_h = dot(light_direction, halfway);

    float energy_bias = mix(0, 0.5, roughness);
    float energy_factor = mix(1.0, 1.0 / 1.51, roughness);
    float fd90 = energy_bias + 2.0 * l_dot_h * l_dot_h * roughness);

    return lambert(diffuse_color) *
        (1.0 + (fd90 - 1.0) * pow(1.0 - n_dot_l, 5.0)) *
        (1.0 + (fd90 - 1.0) * pow(1.0 - n_dot_v, 5.0)) *
        energy_factor;
}
```

Listing 3.3: Frostbite's Disney diffuse BRDF implemented with GLSL.

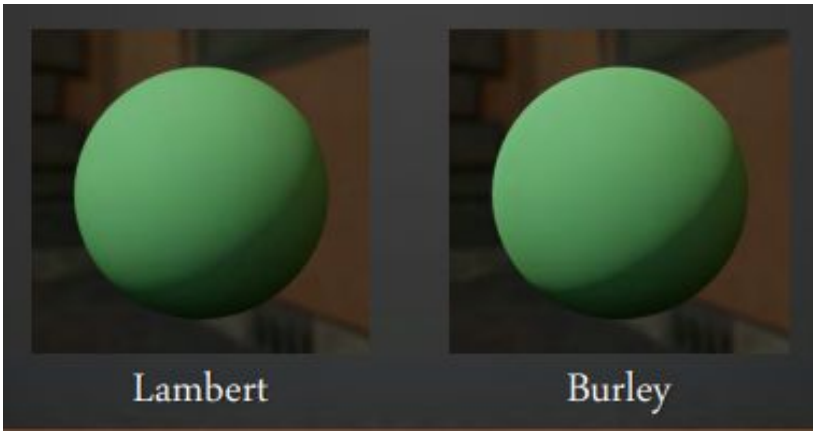


Figure 3.1. Comparison of Lambertian diffuse and Disney diffuse model (Karis, 2013).

3.2. Microfacet Specular BRDF

A general form of the specular microfacet BRDF model for isotropic materials is:

$$f_r(\mathbf{l},\mathbf{v}) = \frac{D(\mathbf{h})F(\mathbf{v},\mathbf{h})G(\mathbf{l},\mathbf{v},\mathbf{h})}{4(\mathbf{n}\cdot\mathbf{l})(\mathbf{n}\cdot\mathbf{v})}. \tag{3.7}$$

This equation is the general *Cook-Torrance* microfacet specular BRDF (Cook and Torrance, 1982). There are three main terms: D is the microfacet distribution function; F is the Fresnel reflection coefficient; and G the geometric attenuation, or shadowing factor. Each term has many different approximations and interpretations which will be explored in this section.

Creating an effective physically-based BRDF requires that these different terms are evaluated.

Both Frostbite and Unreal use a similar microfacet specular BRDF, combining a GGX normal distribution function with a Smith-correlated geometry term and the Fresnel-Schlick approximation (Lagarde and Rousiers, 2014; Karis, 2013). These terms have been researched and have proven to be practical in producing plausible specular reflections. The terms used by Unreal and Frostbite were given more attention than alternative solutions.

3.2.1. Specular D

GGX is the most common NDF used in real-time and offline rendering (Akenine-Möller et al., 2018). Introduced by Trowbridge and Reitz (1975), this distribution function was independently discovered and made famous by Walter et al. (2008) who named it *GGX*. The NDF is also referred to as Trowbridge/Reitz by many papers out of respect for the original authors. The GGX distribution is:

$$D(\mathbf{m}) = \frac{\chi^+(\mathbf{n} \cdot \mathbf{m})\alpha_g^2}{\pi(1 + (\mathbf{n} \cdot \mathbf{m})^2(\alpha_g^2 - 1))^2}. \tag{3.8}$$

Where **m** is a single microfacet normal. Disney exposes the roughness parameter as $\alpha_g = r^2$, where r is the roughness parameter value between 0 and 1 (Burley, 2012). This mapping has been adopted by most applications that use the GGX distribution (Akenine-Möller et al., 2018). Listing 3.4 shows a typical implementation of GGX in GLSL.

```
float d_ggx(vec3 n, vec3 h, float a) {
    float a_squared = a * a;
    float n_dot_h = max(dot(n, h), 0.0);
    float n_dot_h_squared = n_dot_h * n_dot_h;

    float nom = a_squared;
    float denom = (n_dot_h_squared * (a_squared - 1.0) + 1.0);
    denom = PI * denom * denom;

    return nom / denom;
}
```

Listing 3.4: The GGX NDF implemented with GLSL.

3.2.2. Specular G

Heitz (2014) has showed that the only two geometry functions that are valid are the Smith function (Smith, 1967) and the Torrance-Sparrow function (Torrance and Sparrow, 1967), and further clarifies that the Smith function is much closer to the behavior of random microsurfaces (Akenine-Möller et al., 2018). The Smith visibility function Smith's method takes into account both the view direction (geometry obstruction) and the light direction (geometry shadowing):

$$G_2(\mathbf{n}, \mathbf{v}, \mathbf{l}) = G_1(\mathbf{n}, \mathbf{v})G_1(\mathbf{n}, \mathbf{l}). \tag{3.9}$$

There are many forms of the Smith G_1 function. Disney uses the Smith model derived by Walter et al. (2008) for GGX with minor modifications. Lagarde and Rousiers (2014) also favoured this function. Karis (2013) proposed a modification of the Schlick (1994) Smith model, with a minor remapping of the roughness parameter to fit with their use of a GGX NDF:

$$G_1(\mathbf{s}) \approx \frac{\mathbf{n} \cdot \mathbf{s}}{(\mathbf{n} \cdot \mathbf{s})(1 - k) + k}. \tag{3.10}$$

Where **s** can be replaced with either **l** or **v**, and *k* is a remapping of the roughness based on whether the shader is using direct lighting or *image-based lighting* (IBL):

$$k_{direct} = \frac{(\alpha + 1)^2}{8}. \tag{3.11}$$

$$k_{IBL} = \frac{\alpha^2}{2}. \tag{3.12}$$

Image-based lighting is explored further in Section 3.3. This approximation is referred to as Schlick-GGX. Listing 3.5 shows a Schlick-GGX geometry function defined in GLSL.

```
float g_schlick_ggx(float n_dot_v, float k) {
    float nom = n_dot_v;
    float denom = n_dot_v * (1.0 - k) + k;
    return nom / denom;
}

float g_smith(vec3 n, vec3 v, vec3 l, float k) {
    float n_dot_v = max(dot(n, v), 0.0);
    float n_dot_l = max(dot(n, l), 0.0);

    float ggx1 = g_schlick_ggx(n_dot_v, k);
    float ggx2 = g_schlick_ggx(n_dot_l, k);

    return ggx1 * ggx2;
}
```

Listing 3.5: The Schlick-GGX geometry function implemented with GLSL.

3.2.3. Specular F

The Fresnel-Schlick function is an inexpensive approximation that is accurate (Schlick, 1994). Listing 3.6 shows a GLSL implementation of Fresnel-Schlick. It is used by Disney (Burley, 2012) and is used in the Unreal (Karis, 2013) and Frostbite (Lagarde and Rousiers,

2014) game engines. It is simpler than full Fresnel equations, and the error introduced by the approximation is negligible (Burley, 2012). The equation is defined:

$$F(\mathbf{n}, \mathbf{l}) = F_0 + (1 - F_0)(1 - (\mathbf{h} \cdot \mathbf{v}))^5. \tag{3.13}$$

Where F_0 defines the specular reflectance at normal incidence. This must be pre-computed if the same fresnel approximation will be used for both metal and dielectric surfaces. A base reflectivity can be defined that holds for all dielectrics, producing a plausible result without adding complex surface parameters. For Unreal, this value is 0.04 (Karis, 2013). F_0 is linearly interpolated between 0.04 and the albedo using the *metallness* or *metallic* surface property as the interpolant. The *metalness* of a material can be defined as a simple floating point number between 0 for dielectric materials, and 1 for metals (Burley, 2012). This means that if a surface is dielectric, F_0 = 0.04. This technique is possible because metallic surfaces have no diffuse reflections (Akenine-Möller et al., 2018).

```
1  vec3 f0 = vec3(0.04);
2  f0 = mix(f0, surface.rgb, metalness);
3
4  vec3 f_schlick(float cos_theta, vec3 f0) {
5      return f0 + (1.0 - f0) * pow(1.0 - cos_theta, 5.0);
6  }
```

Listing 3.6: The Fresnel-Schlick fresnel approximation implemented with GLSL.

3.3. Image-Based Lighting

Image-based lighting (IBL) is an efficient and effective set of techniques for approximating the incident lighting surrounding a point (Lagarde and Rousiers, 2014; Pranckevičius, 2015). Approximating the integral of incoming light in all directions cannot be achieved in real time; however, it is possible to pre-calculate the integral of incoming light and store it for later use at runtime. When using IBL techniques to solve simulate diffuse and specular effects, environment maps are known as *diffuse light probes* and *specular light probes* (Spogreev, 2016; Akenine-Möller et al., 2018). Bjorke (2004) shows how to achieve localised reflections when using cube maps for image-based lighting, avoiding the typical limitation of cubemap reflections where they appear infinitely distant. A renderer must support HDR imaging to make use of HDR environment maps. The RGBE file format introduced by Ward (1991) allows pixels to have the extended range and precision of floating point values. It can handle bright pixels without loss of precision for darker ones.

3.3.1. Diffuse Light Probes

Cube maps can be used to create diffuse light probes, which store the irradiance of the environment surrounding a point. When used for this purpose, the cube map is called an *irradiance environment map* (Akenine-Möller et al., 2018). Figure 3.2 shows an environment map and an irradiance environment map created from it. Irradiance environment maps are generated by sampling the surrounding lighting environment. The image is then blurred using

image convolution. The extreme blur means that irradiance maps can be stored at low resolution. By storing the convoluted result in each texel of a cube map, sampling the irradiance map in any direction will yield the scene's irradiance in that direction. The surface normal is used to sample the cube map. It is also possible to model the fresnel effect, increasing specular reflectance at glancing angles (Lagarde and Rousiers, 2014).

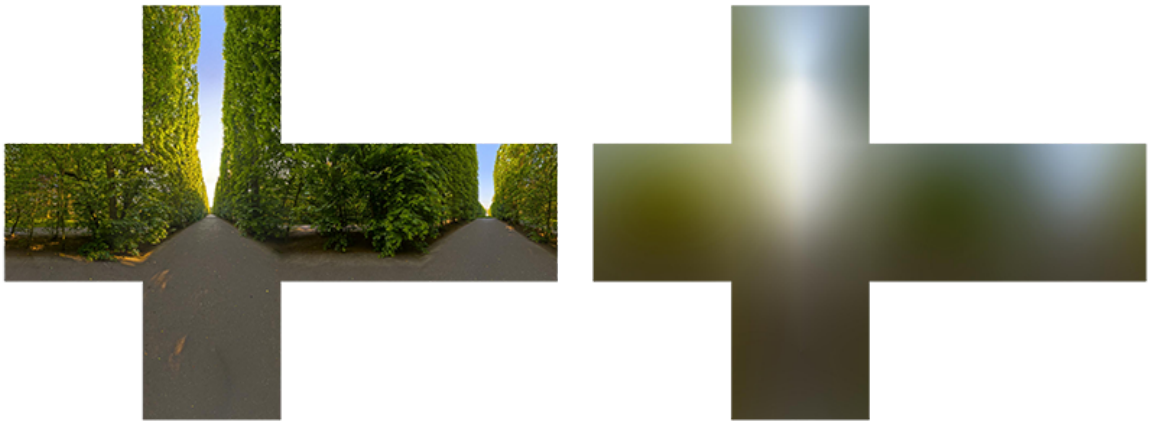


Figure 3.2. Environment map and its resulting irradiance environment map (DeVries, 2016a)

There are many ways to convolute this cube map. A well-known tool to generate probes is AMD CubeMapGen (Lagarde, 2012); however, this tool does all of the filtering work on the CPU, which takes a significant amount of time to process high resolution environment maps. Offloading this work to the GPU can allow this work to be completed in real-time (Spogreev, 2016). For each texel in the environment map, the cosine-weighted contributions of all incident light sources in a normal direction must be summed (Akenine-Möller et al., 2018). An example of this is shown in Figure 3.3.

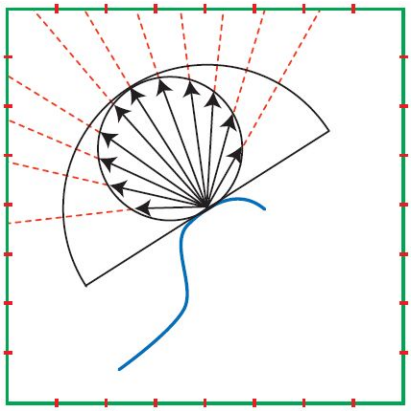


Figure 3.3: Computing an irradiance environment map (Akenine-Möller et al., 2018). The green square denotes the source environment map, while the red marks represent the boundaries between its texels.

Irradiance can be expressed in terms of spherical harmonic coefficients (Ramamoorthi and Hanahan, 2001). King (2005) demonstrates how to do this in real-time on the GPU. Using spherical harmonics as an irradiance map representation is popular as irradiance from

environment lighting is smooth, and the cosine lobe method removes all high-frequency components from the environment map (Akenine-Möller et al., 2018).

3.3.2. Specular Light Probes

Solving the integral of a specular microfacet BRDF for all incoming light positions and including all view directions would be too expensive to perform on a real-time basis. Specular light probes capture the radiance from all directions at a point in the scene in a cube map, also known as a *specular cubemap* (Akenine-Möller et al., 2018). The information stored in the light probe is then used to evaluate BRDFs. Many techniques for pre-convoluting the result of a specular BRDF have been proposed. Karis (2013), Gotanda (2012) and Lazarov (2013) independently derive a similar approximation:

$$\int_{\mathbf{l}\in\Omega} f_r(\mathbf{l},\mathbf{v})L_i(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l} \approx \int_{\mathbf{l}\in\Omega} L_i(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l} * \int_{\mathbf{l}\in\Omega} f_r(\mathbf{l},\mathbf{v})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l}. \quad (3.14)$$

This technique separates the specular work into two parts that can be convoluted then combined to achieve indirect specular image-based lighting. Lagarde and Rousiers (2014) refer to the first term as *LD*, and the second term as *DFG*. LD must be computed for each light probe, whereas DFG can be computed once and reused for all light probes. In this project, LD would be considered the diffuse and specular light probes, while DFG would be considered the BRDF integration map, detailed in Section 3.3.2.2.

3.3.2.1. Pre-Filtered Environment Map

To solve the first half of the equation, a common solution is to use prefiltered environment maps. Similar to the irradiance mapping described in 3.3.1, specular image-based lighting requires an HDR environment map as its convolution input. For increasing roughness levels, the environment map is convoluted with more scattered samples, creating blurred reflections. Each level convoluted is stored in the mip levels of a cube map, decreasing the resolution needed to store each subsequent roughness level (see Figure 3.4).

Figure 3.4. Pre-filtered environment map with filtered roughness levels stored in each mipmap level (DeVries, 2016b)

The diffuse BRDF can be integrated as it relies only on the normal and light directions. Specular BRDFs rely on many variables. Pre-integrating for every view direction and angle of incidence would have a huge memory footprint (Lagarde and Rousiers, 2014). To simplify, **n**, **v**, and the reflected direction **r** are assumed to be equal (Spogreev, 2016). Removing the view dependency allows the specular lobe to be pre-integrated, with the assumption that the BRDF shape is isotropic at all view angles. This compromise prevents introduces error; Figure 3.5 demonstrates how stretched reflections are lost at grazing angles (Karis, 2013; Lagarde and Rousiers, 2014; Spogreev, 2016).

The environment map is convoluted with the GGX distribution using importance sampling (Karis, 2013; Spogreev, 2016). This process is well documented and detailed example code is given by Karis (2013), Lagarde and Rousiers (2014) and Spogreev (2016). *Monte Carlo importance sampling* (Křivánek and Colbert, 2007) captures samples that have a higher influence on the final environment map. For an OpenGL application, this work would be executed in shaders and the final environment map data captured by a framebuffer.

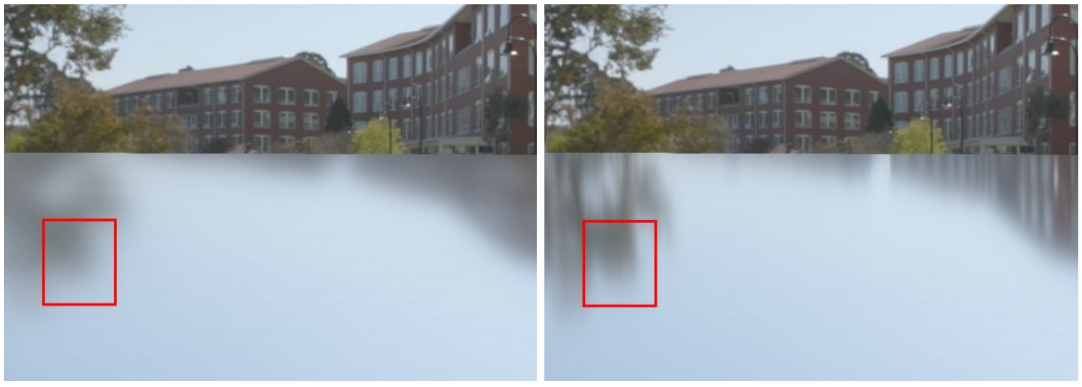


Figure 3.5. Loss of stretched specular reflections at grazing angles (Lagarde and Rousiers, 2014)

3.3.2.2. Environment BRDF

The second integral includes all the lighting information that is missing from the prefiltered specular probe. Karis (2013) describes it as integrating the specular BRDF with a solid white environment. With this assumption, it is possible to pre-calculate the BRDF's response given a roughness value and the angle between the normal and the light direction. Karis stores these values in a *2D lookup texture* (LUT) known as a *BRDF integration map,* which can be seen in Figure 3.6. The LUT outputs a scale (red) and a bias (green) to the surface's Fresnel response, producing the second part of the split specular integral. The Y coordinates map to the roughness, while the X coordinates map to $\mathbf{n} \cdot \mathbf{v}$.

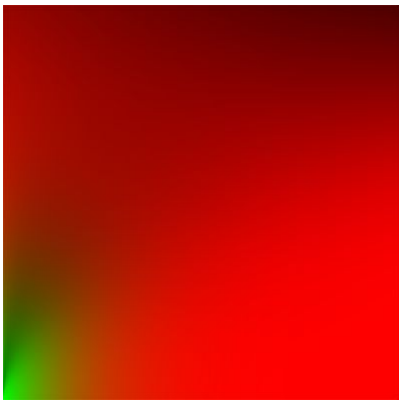


Figure 3.6. BRDF integration map (Karis, 2013)

Karis (2013) provides source code examples on how to generate the LUT. In an OpenGL application, this operation would be performed once by a simple shader, with the result written to a texture. This texture would then be used as a LUT throughout the application.

3.4. Asset Authoring Workflow

PBR not only encompasses a realistic shading model, but also a distinct artist workflow. PBR requires additional material properties to be defined, thus artists need to adjust to a new way of authoring assets. PBR assets require albedo, metallic, and roughness maps to define the material properties (Karis, 2013). It is also typical to see normal maps and ambient occlusion maps being used for PBR materials.

Albedo maps can define the base colour of a non-metallic surface, or the base reflectivity of a metallic surface. This is similar to a diffuse texture that would be used in past ad-hoc models, but albedo maps do not contain any lighting information. Diffuse textures often have shadow detail baked in for contrast - this is unnecessary for PBR. If this detail is needed, ambient occlusion maps can be used. Albedo maps need only the baseline colour, nothing more. Greyscale textures allow material properties to be defined per fragment between 0.0f and 1.0f. Metallic maps define how metallic the material is per texel. Roughness maps allow artists to define how rough the surface is per texel. Ambient Occlusion maps allow artists to add extra shadowing detail to the surface and surrounding geometry. Using AO can show what parts of the material should be in shade, such as the grout between tiles. Figure 3.7 shows a typical set of PBR material properties, and Figure 3.8 shows how these parameters can be combined into a single RGB texture. Normal mapping is not part of PBR but is often used with it to extend the surface detail of materials without introducing additional geometry. Normal maps define the surface normals per-texel, allowing the creation of surface details without extra geometry.

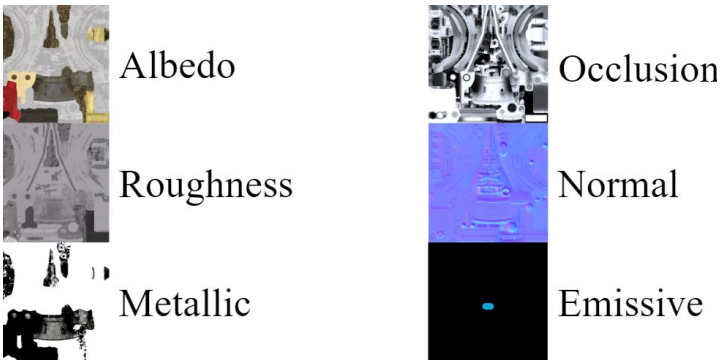


Figure 3.7: A typical set of PBR material properties.

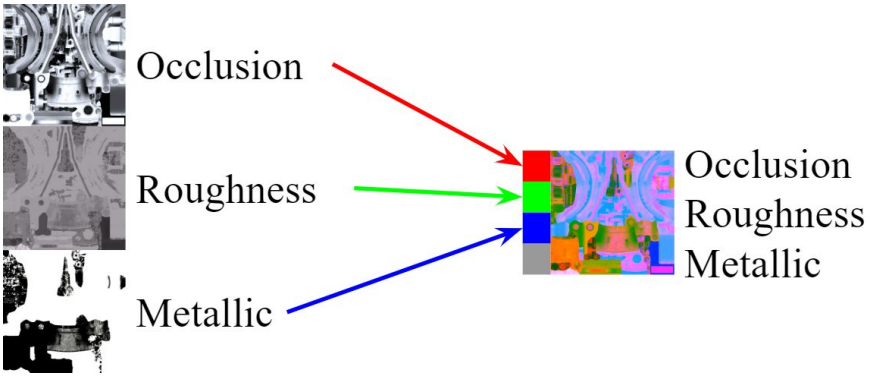


Figure 3.8: Combining the grayscale materials into one texture. Baked ambient occlusion is stored in the red colour component, roughness in the green colour component, and the metal mask in the blue colour component.

4. Software Design

This section details the requirements, development plan, and design of the PBR renderer, *Moka*.

4.1. Development Tools

Moka is developed using C++17, with MSVC and GCC as the main supported compilers. The codebase is designed with extensibility and portability in mind. The project compiles and runs on Windows 10 and Ubuntu with an OpenGL 4.0+ compatible graphics adapter. The graphics system is designed to support multiple API backends, but only an OpenGL backend is implemented. Shaders are written in GLSL. Though spir-v has emerged as a portable intermediate binary shader format capable of being compiled from multiple shading languages, it is only available in the newest versions of OpenGL. Moka only supports a GLSL-based implementation to ensure the project can run on common hardware.

Visual Studio 2017 is the main IDE used for the development of Moka. Visual Studio Code is also used on both Windows and Ubuntu when a full IDE is unnecessary. The project uses CMake, so it is decoupled from any native build system. CMake simplified integrating with third party libraries due to its ubiquity. The project makes extensive use of Git for version control, with the repository hosted on GitHub. Posh Git is used in a Windows development environment to access Git through a Powershell terminal. Downloading, building, and linking to third party libraries is normally a tedious process in C++; but several of package managers have emerged that simplify the process, including Hunter, Conan, and vcPkg. The project makes use of Hunter package manager (Baratov, 2019). ClangFormat is used to enforce a consistent code formatting style (Clang, 2019).

GLM is used as the project's maths library (Riccio, 2018). There would be little benefit from spending time to make a maths library of equivalent depth and functionality; therefore the project committed to GLM as the primary maths library. For windowing and input handling, SDL2 is used (Lantinga, 2018). SDL2 defines a unified C interface to the native windowing systems, sitting on top of Win32 on Windows or X11 on Linux. Though Moka could just as easily implement its own abstractions of these platform-level dependencies, SDL2 is a familiar and well documented library and using it accelerated the development of windowing and input handling. *Spdlog* is used for logging (Melman, 2018). It is a simple, header-only logging library that is highly performant and extensively configurable. *tinyglTF* is a header-only asset importer for glTF model format (Fujita, 2018). Moka makes use of it to import the many test PBR models made available by Khronos Group. *Dear Imgui* is be used for developing simple user interfaces to allow configuration of the rendering environment (Cornut, 2018).

4.2. General Constraints

There are several limitations and constraints to consider that had an effect of the design on the system. Most of these constraints concern hardware and software availability.

Though the software is portable, only Windows / Ubuntu builds are supported. The software only runs on OpenGL version 4.0+ enabled graphics devices. There are also restrictions on the development environment that can compile the software. The machine must have CMake 3.2+ installed to generate solution files. They must also have a C++17-compliant compiler. Only GCC and Visual C++ have been tested. Moka makes use of new language features and libraries that are not yet available in Visual Studio and GCC by default, such as std::filesystem, structured bindings, and if constexpr. The CMakeLists specifies all the compiler arguments necessary to make use of these features.

A significant obstacle was the availability of PBR 3D models. Moka had to import and render PBR assets to prove the effectiveness of the PBR implementation. PBR requires multiple specialised texture maps to define the material properties. Model formats like OBJ and FBX do not define these properties, as they only have standardised properties for more common texture map uses.

A new 3D model format has been established by Khronos Group: glTF 2.0. glTF defines a standardised set of material properties for PBR. It stores geometric data in a binary format that is far faster to import than text-based formats, making it a suitable run-time format for 3D models. glTF is available in most 3D modeling packages, making it easier for artists to create game-ready assets. To support glTF, and to demonstrate its capabilities as a flexible model format, Khronos has made many PBR glTF test models available to the public. This project uses these assets to test PBR rendering. The limited availability of PBR support extends to asset importing libraries. Many popular asset importing libraries have limited or no support for importing PBR materials, and so the same process for preparing assets in non-standard ways has become necessary. As Moka only imports PBR-ready glTF models, there is no reason to commit to a large asset importing library such as Assimp. Moka makes use of tinyglTF, a simple header-only library for reading glTF 2.0 assets.

glTF defines several optional material inputs. Moka copes with this by using shader permutations; when a model is loaded, it can compile a new version of the main PBR shader with all the necessary material inputs. The implementation of this system is described in section 5.2.1.

4.3. Code Style

The style used by the Moka project is derived from Boost. All C++ code produced aims for clarity and correctness, with optimisation as a secondary concern. Moka makes good use of the latest C++ standards, including the use of the C++ standard library where appropriate. Best practices described in Scott Meyers' Effective C++ series are followed (Meyers, 2005; Meyers, 2014). The naming convention of Boost / the C++ standard library is followed:

- All names use `snake_case` except where noted below.
- Acronyms are treated as normal names (`gltf_importer`, not `glTF_importer`).
- Template parameter names begin with an uppercase letter.
- Macro definitions are all uppercase and begin with `MOKA_`.

Clear naming is valued above brevity. Exceptions are used for error reporting where appropriate. Sample programs are provided to demonstrate the use of the library. To ensure consistent formatting; four spaces are used instead of tabs, and line lengths are limited to 80 characters. All source files begin with:

- A comment line referencing the Moka github repository.
- A comment detailing the MIT licence.
- A comment line listing all references used in the source file.

Moka is designed with simplicity in mind. Any features that did not actively contribute towards creating a PBR renderer are considered low priority. Moka is not designed to be the next big game engine, so it is not within the scope of this project to implement a myriad of complicated subsystems. Efforts were focused on creating a simple, working rendering abstraction over complete game engine. This ideal also applied to the design of the user-facing interfaces. Simple and intuitive abstractions are offered to the programmer, allowing them to use PBR with no need to deal with low-level native APIs. The code has a well-documented user-facing API, featuring Doxygen-friendly code comments detailing the responsibilities of each publicly-facing facet of the framework.

4.4. System Design

Moka is a framework for creating physically-based rendering applications. It defines a library offering a high level of abstraction that allows complex PBR rendering to be performed with ease. The application-facing interface is agnostic to any native rendering API / window system and all platform-specific functionality is abstracted. Moka does not provide extensive game engine functionality and is mostly concerned with rendering; though basic input handling logic is provided.

Moka presents an interface for stateless rendering. Unlike OpenGL, any state set for rendering one primitive does not affect subsequent operations. Additionally, draw calls made from the application are not rendered immediately, instead they are added to a buffer that is then processed and rendered later.

Prior to rendering, Moka initializes the native window system; creates a rendering context; initializes the native rendering API; imports PBR-ready 3D assets; and sets up a basic scene. The specific implementation of these features is not relevant to a PBR, so all of this functionality is abstracted and organised into a supporting library of classes. This library is similar in design to MonoGame (2019), presenting useful utilities for quickly creating test applications. The entire application-facing interface is designed in this fashion, providing a high level of abstraction for creating PBR applications. Wherever access to low-level

graphics functionality is exposed, Moka ensures that the platform-specific implementation of these features is well hidden. Larger versions of all UML diagrams can be found in Appendix A.

4.4.1. Application

The architecture of the Moka framework is derivative of existing application frameworks, such as Qt or MonoGame. A central application class is created in the main function and manages the lifetime of the application. Figure 4.1 depicts the application class design.

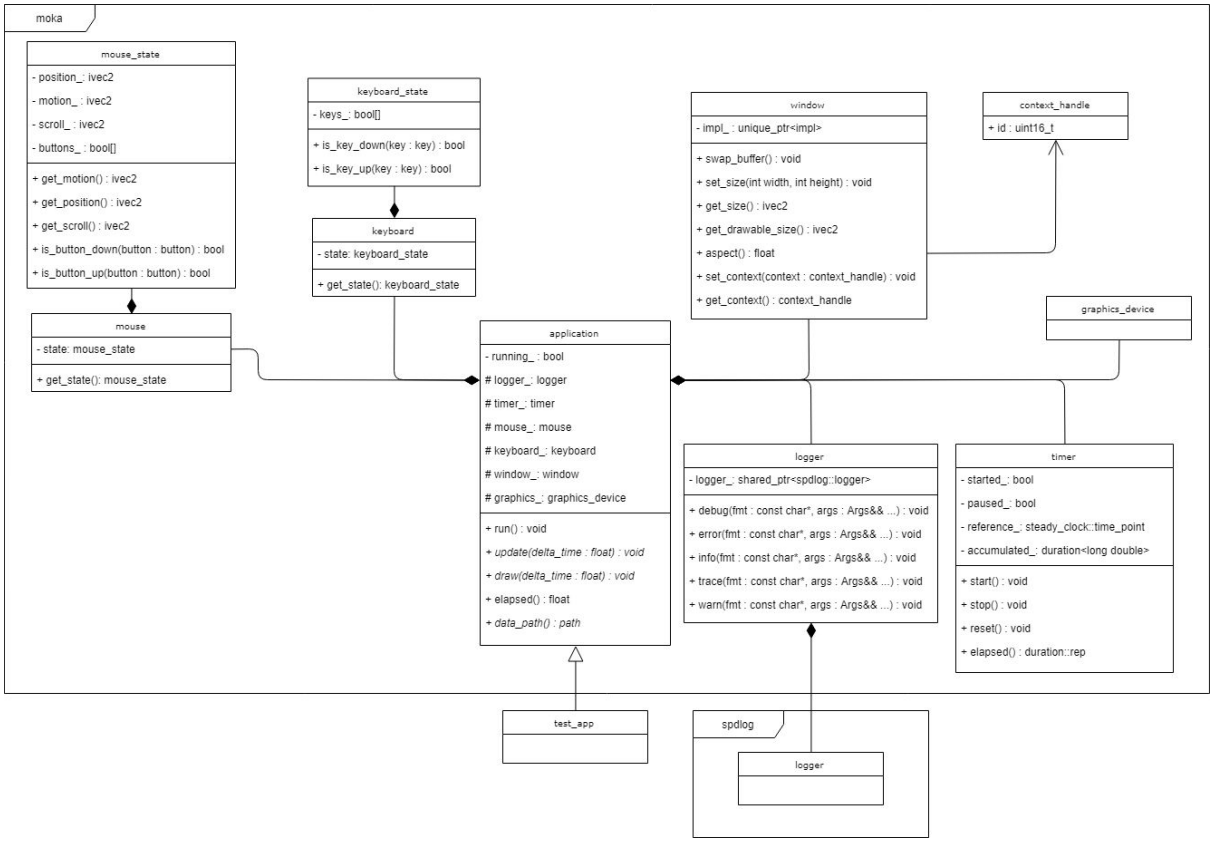


Figure 4.1: UML design of the moka::application class.

Starting a moka application requires programmers to subclass the `application` class. Through `application`, their application has access to the input handling and graphics utilities. Upon calling the `run` method, the game loop starts. The loop calls the virtual `update` and `draw` methods, which must be overloaded in the child class. This is a similar design to XNA / MonoGame's `Game` class (MonoGame, 2019).

The application class abstracts the native event loop from the programmer and updates the input / rendering utilities appropriately. The build system pulls in the correct implementation of the class' functions.

4.4.2. Graphics Device

The Moka `graphics_device` class is an example of a simple *bridge pattern* (Gamma et al., 1994), as the renderer backend is a concrete implementation of an abstract class. The Moka application does not interact with OpenGL directly and is therefore loosely coupled to the native rendering API. This makes Moka extensible, making it entirely possible to write API backends for several target platforms. The graphics device class exposes *builder pattern* (Gamma et al., 1994) classes that make it easier to initialise graphics resources. The application enqueues commands in a `command_list` object before handing them to the `graphics_device` to execute. Each `graphics_command` is an example of the *visitor pattern* (Gamma et al., 1994). Commands can be sorted before being executed using an `uint64_t` id field as the key. When creating a `command_buffer` object, the programmer is given the option of assigning a sort key to the buffer. This allows for sort-based draw call bucketing, a technique for limiting the performance hit of frequent state changes or order-dependent effects. The implementation of this technique is explored in Section 5.2.2. Figure 4.2 shows the design of `graphics_device` in UML.

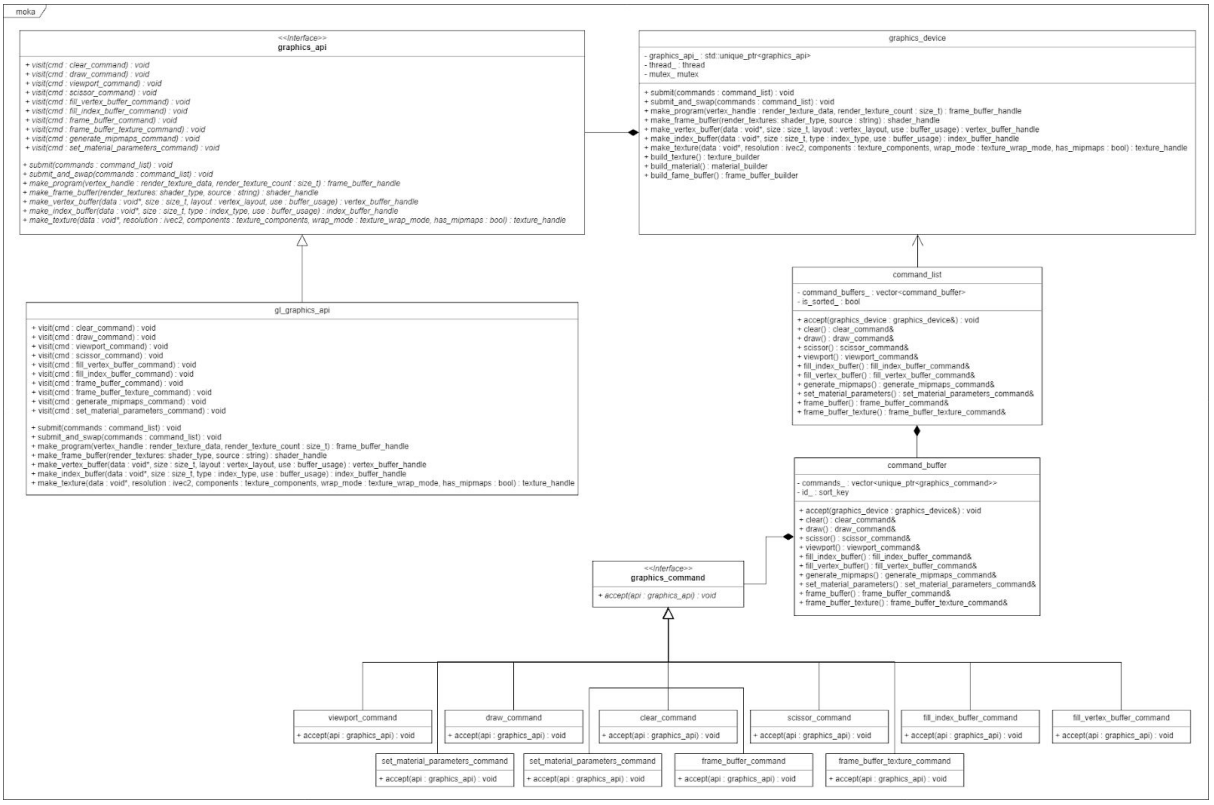


Figure 4.2: UML design of the `graphics_device` class.

4.4.3. Graphics Resources

Many graphics libraries such as MonoGame (2019) define their graphics abstraction through inheritance hierarchies. Moka eschews the typical OOP approach by exposing graphics resources to the application as integer handles. Moka's approach to this problem is inspired largely by BGFX (Karadžić, 2012), a cross-platform C++ rendering library. Instead of creating high-level classes with state and functionality, Moka defines each graphics resource as a struct that encapsulates a 16-bit unsigned integer, as demonstrated in Figure 4.3. It is the

responsibility of the `graphics_api` class to map these integer handles to the low-level native API's resources. By doing this, Moka does not need to worry about leveraging runtime polymorphism. The unique resource handles are small and can be copied without any performance concerns. This design is useful when paired with the graphics commands explored in section 4.4.4, as each graphics command can encapsulate the graphics handles and state it requires to submit to the GPU.

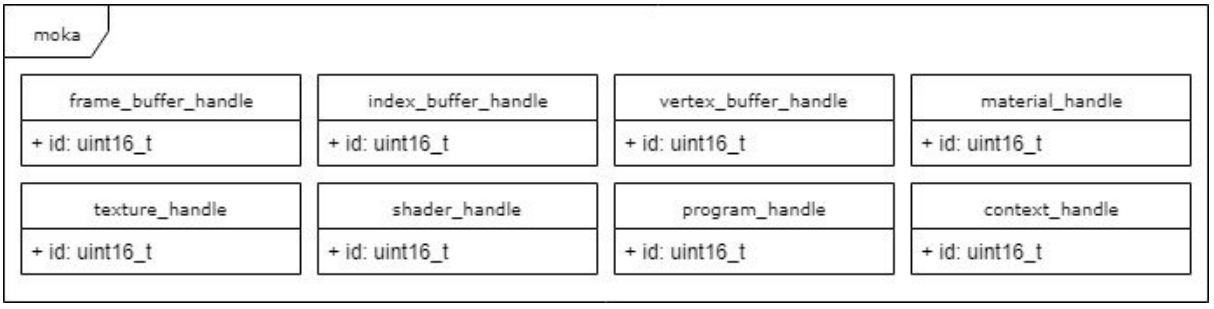


Figure 4.3: UML design of the graphics resource classes. The specific uses of these ids is entirely up to the *graphics_api* implementation. For low-level OpenGL resources such as textures or vertex buffers, ids are cast to GLuint. For higher-level constructs such as materials, the id is an index into an array of shaders, uniforms and rendering settings.

4.4.5. Model System

The model system is loosely based upon the glTF specification. The high-level UML can be seen in Figure 4.4. It evolved over the course of development to accommodate a great number of renderable objects. The `model` class is composed of one or more meshes; the `mesh` class is composed of one or more primitives. The `primitive` class contains a `vertex_buffer_handle`, `index_buffer_handle`, and `material_handle`. The `material_handle` specifies the material that should be used when drawing the primitive. It is an array index which tells `graphics_device` which material to use when rendering the primitive. The `transform` class is general enough to be used for camera transformation. It features helper functions: `look_at`, `rotate_around`, and `to_matrix`.

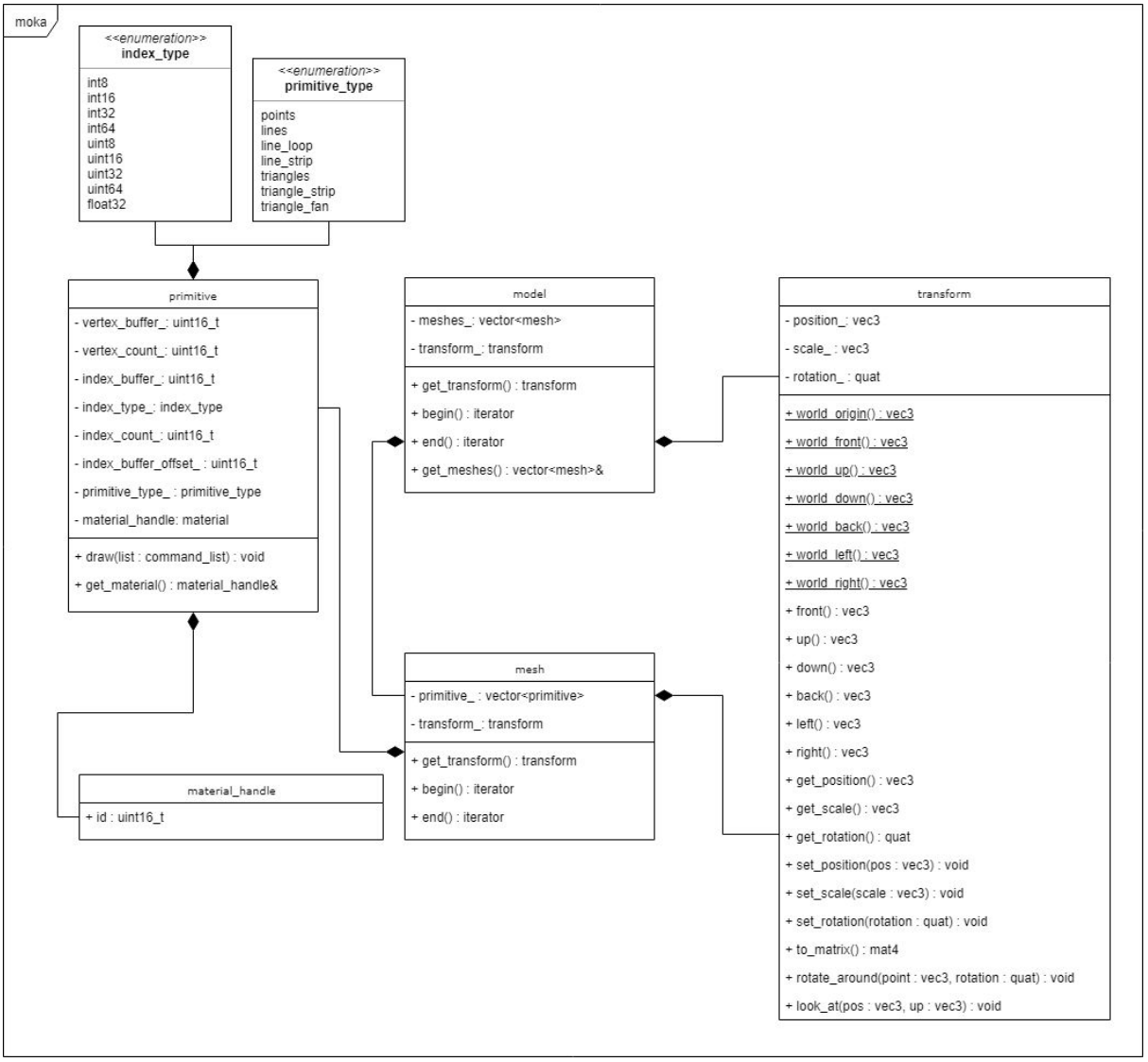


Figure 4.4: UML of a moka::model class.

4.5. Development Methodology

Software development methodologies encourage a systemic approach towards development to improve the quality of the final product. A project of this size and complexity required forward planning to ensure its success. This project was implemented alongside additional research and experimentation, so it was entirely possible that the project requirements or implementation plans could have changed because of information discovered during development. This means that a traditional linear project management methodology, such as the Waterfall model, was unsuitable. Rewriting the project schedule to compensate for new obstacles would have been very time consuming, which was unacceptable given the time constraints the project already faced.

While the project did not benefit from a rigid linear project plan, deadlines still needed to be met. Moka was developed using a Scrum-based methodology. Scrum is an agile methodology that promotes a clear project schedule, but also allows the developer to respond to change during the development process (Schwaber and Sutherland, 2016). Development was broken into discrete tasks that could be completed over a period of 2 week ‘sprints’. To ensure that the project was progressing as planned, the project was reassessed at the end of each sprint. It was at this time that the project supervisor was consulted and the progress made and work remaining was considered.

5. Implementation

In this section, the implementation of Moka's physically-based rendering techniques is explored.

5.1. PBR

5.1.1. Image-Based Lighting

The majority of the techniques employed in this project revolve around pre-calculating lighting information and storing the results in a cubemap. Much of the code for rendering to a cubemap was exactly the same, therefore it was refactored out into two reusable utility functions, detailed in Listing 5.1 and Listing 5.2.

```
void pbr_util::draw_to_cubemap(
    int size,
    int mip_level,
    texture_handle cubemap,
    material_handle material,
    draw_callback&& pre,
    draw_callback&& post) const {

    command_list list;

    const auto hdr_frame_buffer =
        device_.build_frame_buffer()
            .add_depth_attachment(frame_format::depth_component24, size, size)
            .build();

    list.viewport()
        .set_rectangle(0, 0, size, size);

    list.frame_buffer()
        .set_frame_buffer(hdr_frame_buffer);

    if (pre) {
        pre(list);
    }

    draw_cubemap_faces(list, mip_level, cubemap, material);

    if (post) {
        post(list);
    }

    list.frame_buffer().
        set_frame_buffer({0});

    device_.submit(std::move(list), false);
    device_.destroy(hdr_frame_buffer);
}
```

Listing 5.1: A utility for rendering to a cubemap. Defined in *engine/src/graphics/pbr.cpp*.

In Listing 5.1, the function takes a size, mip level, and material. The material defines all the technique-specific shader logic, which is then used to shade a cube model. Line 23 calls an optional `pre` function allowing the calling code to register extra rendering commands that can be submitted immediately before rendering. Line 26 calls the function `draw_cubemap_faces`, which performs the actual rendering and is detailed in Listing 5.2. Line 29 of Listing 5.1 calls an optional `post` function that allows the calling code to register any extra rendering commands that should be submitted after rendering. This function is used by all of the environment mapping techniques in this section.

```
1  void pbr_util::draw_cubemap_faces(
2      command_list& list,
3      int mip_level,
4      texture_handle cubemap,
5      material_handle material) const {
6
7      for (auto i = 0; i < 6; i++) {
8          list.set_material_parameters()
9              .set_material(material)
10             .set_parameter("view", constants::capture_views[i]);
11
12         const auto image_target = constants::image_targets[i];
13
14         list.frame_buffer_texture()
15             .set_texture(cubemap)
16             .set_attachment(frame_attachment::color)
17             .set_target(image_target)
18             .set_mip_level(mip_level);
19
20         list.clear()
21             .set_clear_color(true)
22             .set_clear_depth(true);
23
24         list.draw()
25             .set_vertex_buffer(cube_buffer_)
26             .set_vertex_count(36)
27             .set_primitive_type(primitive_type::triangles)
28             .set_material(material);
29     }
30 }
```

Listing 5.2: A utility for rendering to the faces of a cubemap. Defined in *engine/src/graphics/pbr.cpp*.

In Listing 5.2, the loop starting on line 7 demonstrates how the cube model is captured from 6 angles. Two constant arrays, `capture_views` and `image_targets`, are defined to select the correct view matrix and cubemap target for each face in the cubemap. Each face is rendered with a 90 degree perspective and the final result is captured in the corresponding mip level and texture target of a cubemap.

5.1.1.1. Creating an HDR Environment Map

Most HDR environment maps are distributed as .HDR files, an equirectangular format depicting a 360° view of the environment. To use these assets, Moka defines a helper function to translate equirectangular environment maps into cubemaps. Figure 5.1 depicts an equirectangular environment map.

Figure 5.1: An HDRI environment map (Zaal, 2019).

Moka defines a vertex and fragment shader to help with this operation. These shaders operate on the equirectangular texture map and a vertex buffer that defines a cube. The vertex shader passes the position of the vertex to the fragment shader. The fragment shader is more complicated, and can be seen in Listing 5.3. It shades the cube as though the equirectangular map has been folded onto the sides of the cube. The local position is normalised and used as the sample direction. The direction is then transformed into UV coordinates. The `inv_atan` constant is the multiplicative inverse of 2π and π. The `sample_spherical_map` function on line 6 moves the sample direction from cartesian coordinates to polar angles, before remapping them to be used as uvs. Multiplying the direction by `inv_atan` transformers the values into the -0.5 to 0.5 range. The shader then adds 0.5, remapping the coordinates to 0 to 1. These coordinates can then be used to sample the texture as though it were a cubemap.

```
1  out vec4 frag_color;
2  in vec3 local_pos;
3  uniform sampler2D map;
4  const vec2 inv_atan = vec2(0.1591, 0.3183);
5
6  vec2 sample_spherical_map(vec3 v) {
7      return (vec2(atan(v.z, v.x), asin(v.y)) * inv_atan) + 0.5;
8  }
9
10 void main() {
11     vec2 uv = sample_spherical_map(normalize(local_pos));
12     vec3 color = texture(map, uv).rgb;
13     frag_color = vec4(color, 1.0);
14 }
```

Listing 5.3: Sampling an equirectangular map. Defined in *engine/includes/graphics/default_shaders.hpp*.

Using this shader on a cube model maps the image onto the surface of the cube. These shaders are set up as part of a material which is then passed to `draw_to_cubemap`. Listing 5.4 demonstrates how to use this shader to project an equirectangular map onto a cubemap. A post-render callback is registered on line 20 to generate mipmaps for the cubemap after

rendering. Once this function is complete, it returns a cubemap version of the `equirectangular_map` argument.

```
texture_handle pbr_util::equirectangular_to_cubemap(
    texture_handle equirectangular_map, const int environment_size) const {

    const auto hdr_material =
        device_.build_material()
            .set_vertex_shader(shaders::equirectangular_to_cube::vert)
            .set_fragment_shader(shaders::equirectangular_to_cube::frag)
            .add_material_parameter("projection", constants::projection)
            .add_material_parameter("view", glm::mat4{})
            .add_material_parameter("map", equirectangular_map)
            .set_culling_enabled(false)
            .build();

    const auto hdr_cubemap = make_empty_hdr_cubemap(
        environment_size, min_filter::linear_mipmap_linear, false);

    draw_to_cubemap(
        environment_size, 0, hdr_cubemap, hdr_material,
        {},
        [&](command_list& list) {
            list.generate_mipmaps().set_texture(hdr_cubemap);
        });

    return hdr_cubemap;
}
```

Listing 5.4: Creating a cubemap from an equirectangular map. Defined in *engine/src/graphics/pbr.cpp*.

If `equirectangular_to_cubemap` is called with the equirectangular environment map from Figure 5.1, it produces the cube mapped version depicted in Figure 5.2.

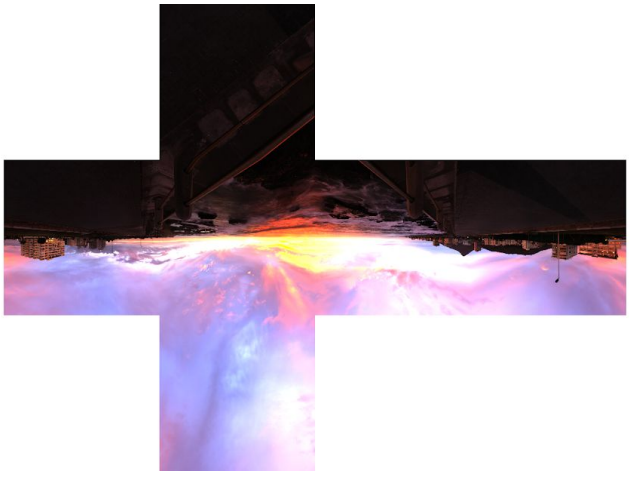


Figure 5.2, a cubemap version of Figure 5.1 generated by Moka.

5.1.1.2. Creating Diffuse Light Probes

As defined in section 3.3.1, the goal of irradiance environment mapping is to solve the pre-integrate all diffuse lighting. There are many possible implementations of this

convolution. Moka's solution approximates a solution to the integral by taking a fixed amount of discrete samples and averaging the results. This can be seen in Listing 5.5. The samples are taken along a hemisphere that is oriented around the sample direction. Each sample is uniformly spread within the hemisphere. Increasing the number of samples should improve the results as it more closely approaches a perfect solution to the integral. Due to the highly blurred image produced by this function, the cubemap can be generated with a modest 32 * 32 resolution for each face.

```
1  out vec4 fragment_color;
2  in vec3 world_position;
3  uniform samplerCube environment_map;
4  const float PI = 3.14159265359;
5
6  void main()
7  {
8      vec3 n = normalize(world_position);
9      vec3 irradiance = vec3(0.0);
10     vec3 up    = vec3(0.0, 1.0, 0.0);
11     vec3 right = cross(up, n);
12     up         = cross(n, right);
13     float sample_delta = 0.025;
14     float number_of_samples = 0.0;
15
16     for(float phi = 0.0; phi < 2.0 * PI; phi += sample_delta)
17     {
18         for(float theta = 0.0; theta < 0.5 * PI; theta += sample_delta)
19         {
20             vec3 tangent_sample = vec3(sin(theta) * cos(phi), sin(theta) * sin(phi), cos(theta));
21             vec3 sample_vec = tangent_sample.x * right + tangent_sample.y * up + tangent_sample.z * n;
22             irradiance += texture(environment_map, sample_vec).rgb * cos(theta) * sin(theta);
23             number_of_samples++;
24         }
25     }
26     irradiance = PI * irradiance * (1.0 / float(number_of_samples));
27
28     fragment_color = vec4(irradiance, 1.0);
29 }
```

Listing 5.5: Prefiltering a diffuse light probe. Defined in *engine/src/graphics/default_shaders.hpp.*

In listing 5.5, a `sample_delta` variable is defined on line 13 to control the level of detail when traversing the hemisphere. If this value is made smaller, the number of samples taken increases. The for loop on line 16 allows the shader to integrate over the polar coordinates `phi` and `theta`, which are then converted into line 20's cartesian sample vector, `tangent_sample`. This sample vector is defined in tangent space, so it is then converted into world space, `sample_vec`. This variable is used to sample the environment map. On line 22, the sample is added to the total `irradiance`. At the end of the loop, the irradiance is divided by the number of samples taken on line 26, giving the average irradiance.

Listing 5.6 demonstrates how this shader can be used in a material and passed to `draw_to_cubemap`. If the environment map from Figure 5.2 is passed to this function, it produces the cubemap depicted in Figure 5.3.

```
texture_handle pbr_util::make_irradiance_environment_map(texture_handle hdr_environment_map) const
{
    const auto irradiance_material =
        device_.build_material()
            .set_vertex_shader(shaders::cubemap::vert)
            .set_fragment_shader(shaders::make_irradiance_map::frag)
            .add_material_parameter("projection", constants::projection)
            .add_material_parameter("view", glm::mat4{})
            .add_material_parameter("environment_map", hdr_environment_map)
            .set_culling_enabled(false)
            .build();

    const auto irradiance_size = 32;

    const auto irradiance_cubemap =
        make_empty_hdr_cubemap(irradiance_size, min_filter::linear, false);

    draw_to_cubemap(irradiance_size, 0, irradiance_cubemap, irradiance_material);

    return irradiance_cubemap;
}
```

Listing 5.6: Creating an irradiance environment map. Defined in *engine/src/graphics/pbr.cpp*.

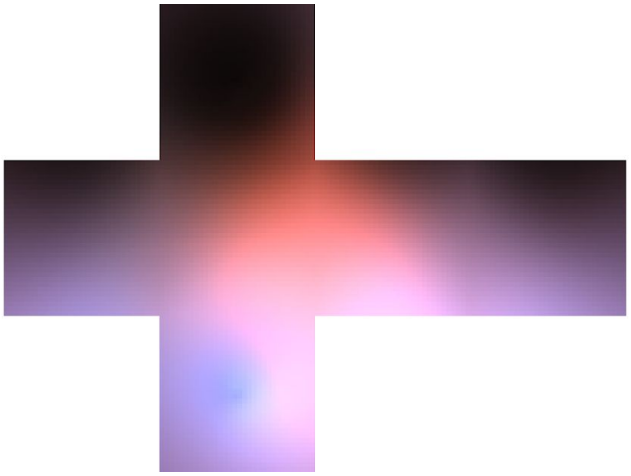


Figure 5.3, an irradiance environment map generated from Figure 5.2 by Moka.

5.1.1.3. Creating Specular Light Probes

Prefiltering the specular environment map is a similar process to prefiltering the irradiance environment map. The roughness of the surface is used to convolute the image before storing the final results in the mip levels of the cubemap. Rough surfaces require blurred images to mimic the scattered reflection, so the rougher images are stored in the lower resolution mip levels. This shader is derived largely from the code samples provided by Karis (2013) and Lagarde and Rousiers (2014). The shader can be seen in Listing 5.7.

```
void main() {
    // https://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf
    // https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf
    vec3 n = normalize(world_position);
    vec3 r = n;
    vec3 v = r;

    const uint sample_count = 2048u;
    vec3 prefiltered_color = vec3(0.0);
    float total_weight = 0.0;

    for(uint i = 0u; i < sample_count; ++i) {
        vec2 xi = hammersley(i, sample_count);
        vec3 h = importance_sample_ggx(xi, n, roughness);
        vec3 l  = normalize(2.0 * dot(v, h) * h - v);
        float n_dot_l = max(dot(n, l), 0.0);

        if(n_dot_l > 0.0) {
            float d   = distribution_ggx(n, h, roughness) / PI;
            float n_dot_h = max(dot(n, h), 0.0);
            float h_dot_v = max(dot(h, v), 0.0);
            float pdf = d * n_dot_h / (4.0 * h_dot_v) + 0.0001;
            float resolution = 1024.0;
            float sa_texel  = 4.0 * PI / (6.0 * resolution * resolution);
            float sa_sample = 1.0 / (float(sample_count) * pdf + 0.0001);
            float mip_level = roughness == 0.0 ? 0.0 : 0.5 * log2(sa_sample / sa_texel);
            prefiltered_color += textureLod(environment_map, l, mip_level).rgb * n_dot_l;
            total_weight      += n_dot_l;
        }
    }

    if(total_weight > 0.0) {
        prefiltered_color = prefiltered_color / total_weight;
    }

    fragment_color = vec4(prefiltered_color, 1.0);
}
```

Listing 5.7: Pre-filtering a specular light probe. Defined in *engine/src/graphics/default_shaders.hpp.*

This shader pre-calculates the specular portion of the reflectance equation using importance sampling. Like the irradiance convolution, the process consists of a main loop which calculates a set number of sample directions that are then averaged at the end. A low-discrepancy sequence is used to generate the sample direction. Moka cannot afford to evaluate the integral for all incident light directions, so it instead samples a set number of random directions and averages the results, producing an approximation of the integral (Křivánek and Colbert, 2007). The above shader uses the Hammersley Sequence (Spogreev, 2016), a random low-discrepancy sequence that can generate hemisphere directions in real-time. Listing 5.8 shows Moka's implementation of the Hammersley sequence.

```
float radical_inverse_vdc(uint bits) {
     bits = (bits << 16u) | (bits >> 16u);
     bits = ((bits & 0x55555555u) << 1u) | ((bits & 0xAAAAAAAAu) >> 1u);
     bits = ((bits & 0x33333333u) << 2u) | ((bits & 0xCCCCCCCCu) >> 2u);
     bits = ((bits & 0x0F0F0F0Fu) << 4u) | ((bits & 0xF0F0F0F0u) >> 4u);
     bits = ((bits & 0x00FF00FFu) << 8u) | ((bits & 0xFF00FF00u) >> 8u);
     return float(bits) * 2.3283064365386963e-10;
}

vec2 hammersley(uint i, uint n) {
    return vec2(float(i) / float(n), radical_inverse_vdc(i));
}
```

Listing 5.8: The Hammersley sequence defined in GLSL. Can be found in *engine/src/graphics/default_shaders.hpp.*

Listing 5.8 makes use of a common implementation that uses bitwise operations to accelerate the calculation (Dammertz, 2012). In Listing 5.7, calling the `hammersley` function returns the random sample direction. Once the initial sample direction is generated, the shader orients and biases the sample towards the specular lobe. Karis (2013) defines an importance sampling function for this purpose, which is implemented as `importance_sample_ggx` in Listing 5.9. This same function is used by Frostbite (Lagarde and Rousiers, 2014) and EA Sports' *Ignite* engine (Spogreev, 2016). In this function, a sample vector is calculated oriented around the microsurface halfway vector based on the roughness parameter and the low-discrepancy sequence value `xi`. This implementation accepts Burley's parameterisation of perceptual roughness as the roughness parameter squared (Burley, 2012).

```
vec3 importance_sample_ggx(vec2 xi, vec3 n, float roughness) {
    // https://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf
    // Spogreev, I. (2016). Physically Based Light Probe Generation on GPU.
    float a = roughness * roughness;

    float phi = 2.0 * PI * xi.x;
    float cos_theta = sqrt((1.0 - xi.y) / (1.0 + (a*a - 1.0) * xi.y));
    float sin_theta = sqrt(1.0 - cos_theta * cos_theta);

    vec3 h;
    h.x = cos(phi) * sin_theta;
    h.y = sin(phi) * sin_theta;
    h.z = cos_theta;
    h = normalize(h);

    vec3 up        = abs(n.z) < 0.999 ? vec3(0.0, 0.0, 1.0) : vec3(1.0, 0.0, 0.0);
    vec3 tangent   = normalize(cross(up, n));
    vec3 bitangent = cross(n, tangent);

    vec3 sample_vec = tangent * h.x + bitangent * h.y + n * h.z;
    return normalize(sample_vec);
}
```

Listing 5.9: GGX importance sampling. Defined in *engine/src/graphics/default_shaders.hpp.*

On line 33 of Listing 5.7, the final colour is divided against the total sample weight, which is the sum of all `n_dot_l` values for the samples. Samples that contribute less to the final result have a small `n_dot_l` value, and therefore contribute less to the final weight. The resulting colour that is sampled is summed with the `prefiltered_color` variable. The

`prefiltered_color` variable is divided by the sample weight. The environment is prefiltered based on a roughness parameter that varies over the mipmap levels of the cubemap. With the shader defined, the rest of the work is to capture the results in the mip levels of a cubemap, as shown in Listing 5.10. As demonstrated in sections 5.1.1.1 and 5.1.1.2, it is possible to set up this work as a material and pass it to the `draw_to_cubemap` function, specifying the appropriate mip levels of the cubemap that should be rendered to. A pre-render command is also passed to the function as a lambda. It injects a command that sets the correct roughness parameter for shading. If `make_specular_environment_map` is invoked with the cubemap depicted in Figure 5.2, it produces the cubemap depicted in Figure 5.4. Notice that each mip level contains an increasingly convoluted version of the image.

```
texture_handle pbr_util::make_specular_environment_map(texture_handle hdr_environment_map) const {
    const auto prefilter_size = 256;

    const auto prefilter_cubemap = make_empty_hdr_cubemap(
        prefilter_size, min_filter::linear_mipmap_linear, true);

    const auto prefilter_material =
        device_.build_material()
            .set_vertex_shader(shaders::make_specular_map::vert)
            .set_fragment_shader(shaders::make_specular_map::frag)
            .add_material_parameter("roughness", 0.0f)
            .add_material_parameter("environment_map", hdr_environment_map)
            .add_material_parameter("projection", constants::projection)
            .add_material_parameter("view", glm::mat4{})
            .set_culling_enabled(false)
            .build();
    const size_t max_mip_levels = 5;

    for (size_t mip = 0; mip < max_mip_levels; ++mip) {
        const auto mip_size = size_t(prefilter_size * std::pow(0.5, mip));

        draw_to_cubemap(
            mip_size, mip, prefilter_cubemap, prefilter_material, [&](command_list& list) {
                const auto roughness = static_cast<float>(mip) /
                                       static_cast<float>(max_mip_levels - 1);
                list.set_material_parameters()
                    .set_material(prefilter_material)
                    .set_parameter("roughness", roughness);
            });
    }
    return prefilter_cubemap;
}
```

Listing 5.10: Creating a specular environment map. Defined in *engine/src/graphics/pbr.cpp*.

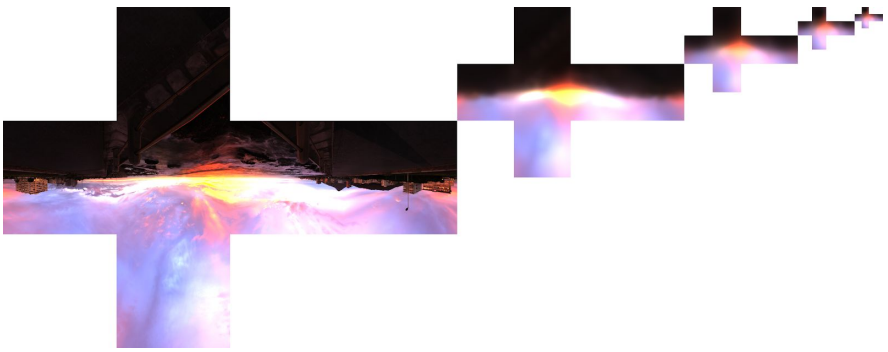


Figure 5.4, a specular environment map generated from Figure 5.2 by Moka.

5.1.1.4. Creating the BRDF Integration Map

Karis (2013) defines how to calculate the BRDF integration map. Listing 5.11 demonstrates the implementation, which is based heavily on the code provided by Karis.

```
1  vec2 integrate_brdf(float n_dot_v, float roughness) {
2      // https://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf
3      vec3 v;
4      v.x = sqrt(1.0 - n_dot_v * n_dot_v);
5      v.y = 0.0;
6      v.z = n_dot_v;
7
8      float a = 0.0;
9      float b = 0.0;
10     vec3 n = vec3(0.0, 0.0, 1.0);
11
12     const uint sample_count = 2048u;
13
14     for(uint i = 0u; i < sample_count; ++i)
15     {
16         vec2 xi = hammersley(i, sample_count);
17         vec3 h = importance_sample_ggx(xi, n, roughness);
18         vec3 l = normalize(2.0 * dot(v, h) * h - v);
19
20         float n_dot_l = max(l.z, 0.0);
21         float n_dot_h = max(h.z, 0.0);
22         float v_dot_h = max(dot(v, h), 0.0);
23
24         if(n_dot_l > 0.0)
25         {
26             float g = geometry_smith(n, v, l, roughness);
27             float g_vis = (g * v_dot_h) / (n_dot_h * n_dot_v);
28             float fc = pow(1.0 - v_dot_h, 5.0);
29
30             a += (1.0 - fc) * g_vis;
31             b += fc * g_vis;
32         }
33     }
34     a /= float(sample_count);
35     b /= float(sample_count);
36     return vec2(a, b);
37 }
```

Listing 5.11: Sampling a BRDF integration map. Defined in *engine/src/graphics/default_shaders.hpp.*

As discussed in section 3.2.2, the geometry term uses a different *k* term when being used for image-based lighting. The implementation of the `geometry_smith` function used with Listing 5.11 on line 26 can be seen in Listing 5.12.

```
1  float geometry_schlick_ggx(float n_dot_v, float roughness) {
2      float a = roughness;
3      float k = (a * a) / 2.0;
4
5      float numerator   = n_dot_v;
6      float denominator = n_dot_v * (1.0 - k) + k;
7
8      return numerator / denominator;
9  }
10
11 float geometry_smith(vec3 n, vec3 v, vec3 l, float roughness) {
12     float n_dot_v = max(dot(n, v), 0.0);
13     float n_dot_l = max(dot(n, l), 0.0);
14     float ggx2 = geometry_schlick_ggx(n_dot_v, roughness);
15     float ggx1 = geometry_schlick_ggx(n_dot_l, roughness);
16
17     return ggx1 * ggx2;
18 }
```

Listing 5.12: Geometry term for use in IBL. Defined in *engine/src/graphics/default_shaders.hpp.*

In Listing 5.11, the viewing angle is taken as an input alongside the roughness parameter. On lines 16 and 17, a sample direction is again calculated using `hammersley` and then biased using `importance_sample_ggx`. The sample direction is then processed over the geometry and fresnel term of the BRDF, creating a scale and bias to F_0 for each sample before averaging them. Listing 5.13 demonstrates how to use this shader to create the BRDF integration map.

```
texture_handle pbr_util::make_brdf_integration_map() const {
    const auto brdf_material =
        device_.build_material()
            .set_vertex_shader(shaders::make_brdf_map::vert)
            .set_fragment_shader(shaders::make_brdf_map::frag)
            .build();

    command_list brdf_list;
    const auto brdf_size = 512;
    const auto brdf_image = device_.build_texture()
                                .add_image_data(
                                    image_target::texture_2d,
                                    0,
                                    device_format::rg16f,
                                    brdf_size,
                                    brdf_size,
                                    0,
                                    host_format::rg,
                                    pixel_type::float32,
                                    nullptr)
                                .set_target(texture_target::texture_2d)
                                .set_wrap_s(wrap_mode::clamp_to_edge)
                                .set_wrap_t(wrap_mode::clamp_to_edge)
                                .set_min_filter(min_filter::linear)
                                .set_mag_filter(mag_filter::linear)
                                .build();

    const auto brdf_frame_buffer =
        device_.build_frame_buffer()
            .add_depth_attachment(frame_format::depth_component24, brdf_size, brdf_size)
            .build();

    brdf_list.frame_buffer().set_frame_buffer(brdf_frame_buffer);

    brdf_list.viewport().set_rectangle(0, 0, brdf_size, brdf_size);

    brdf_list.frame_buffer_texture()
        .set_texture(brdf_image)
        .set_attachment(frame_attachment::color)
        .set_target(image_target::texture_2d)
        .set_mip_level(0);

    brdf_list.clear().set_clear_color(true).set_clear_depth(true);

    brdf_list.draw()
        .set_vertex_buffer(quad_buffer_)
        .set_vertex_count(4)
        .set_primitive_type(primitive_type::triangle_strip)
        .set_material(brdf_material);

    brdf_list.frame_buffer().set_frame_buffer({0});

    device_.submit(std::move(brdf_list), false);

    device_.destroy(brdf_frame_buffer);

    return brdf_image;
}
```

Listing 5.13: Creating a BRDF integration map. Defined in *engine/src/graphics/pbr.cpp*.

The `make_brdf_integration_map` function is very similar to Listings 5.4, 5.6 and 5.10. with the difference being that it operates on a two dimensional texture. Invoking `make_brdf_integration_map` returns the texture depicted in Figure 3.6.

5.1.2. Main PBR Shader

The main PBR shader samples the material's surface parameters, combines the different parts of the split-sum approximation and uses them to calculate the final output colour of the fragment. The main function of the PBR shader can be seen in Listing 5.14.

```
1  void main() {
2      // https://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf
3      // https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf
4      if(discard_fragment()) {
5          discard;
6      }
7
8      // fetch the material properties
9      vec4 albedo = get_albedo();
10     vec4 emissive = get_emissive();
11     float metallic = get_metallic();
12     float roughness = get_roughness();
13     float ao = get_ao();
14
15     // fetch input lighting data
16     vec3 n = get_normal();
17     vec3 v = normalize(view_pos - in_frag_pos);
18     vec3 r = reflect(-v, n);
19
20     // get the diffuse light contribution
21     vec4 irradiance = texture(irradiance_map, n);
22     vec4 diffuse = irradiance * albedo;
23
24     // calculate the reflectance at normal incidence
25     vec3 f0 = vec3(0.04);
26     f0 = mix(f0, albedo.rgb, metallic);
27     vec3 kS = fresnel_schlick_roughness(max(dot(n, v), 0.0), f0, roughness);
28     vec3 kD = 1.0 - kS;
29     kD *= 1.0 - metallic;
30
31     // calculate final lighting
32     vec3 prefiltered_color = textureLod(prefilter_map, r, roughness * mip_count).rgb;
33     vec2 brdf  = texture(brdf_lut, vec2(max(dot(n, v), 0.0), roughness)).rg;
34     vec4 specular = vec4(prefiltered_color * (kS * brdf.x + brdf.y), 1.0);
35     vec4 ambient = (vec4(kD, 1.0) * diffuse + specular) * ao;
36     vec4 color = ambient + emissive;
37
38     // gamma correction + HDR tone mapping
39     vec4 mapped = vec4(1.0) - exp(-color * exposure);
40     vec3 corrected_color = vec3(1.0 / gamma);
41     frag_color = vec4(pow(mapped.xyz, corrected_color), albedo.a);
42 }
```

Listing 5.14: The main PBR shader code. Defined in *examples/assets/Materials/Shaders/pbr.frag*.

On line 4 of Listing 5.14, the shader decides if the fragment should be discarded. glTF assets require this feature. When the `MASK_ALPHA` preprocessor definition is present, the `material.alpha_cutoff` uniform defines the cutoff threshold. The implementation of `discard_fragment` can be seen in Listing 5.15.

```
1  bool discard_fragment() {
2      #ifdef MASK_ALPHA
3          return texture(material.diffuse_map, in_texture_coord).a < material.alpha_cutoff;
4      #else
5          return false;
6      #endif
7  }
```

Listing 5.15: Discarding a fragment when masking is required. Defined in *examples/assets/Materials/Shaders/pbr.frag*.

Figure 5.5 shows an example of why this is necessary. Without alpha masking, some transparent textures become erroneously opaque. Alpha masking discards the fragment when appropriate before any shading is performed.

Figure 5.5: Shading Sponza without (left) and with (right) alpha masking enabled.

In lines 9 - 13 of Listing 5.14, the shader retrieves the material's surface properties. Moka defines a preprocessor-driven system for dealing with shader permutations. The CPU-side implementation of this feature is detailed in Section 5.2.1. A selection of helper functions are defined to encapsulate the preprocessor usage.

```
vec4 get_albedo() {
    #ifdef DIFFUSE_MAP
        return texture(material.diffuse_map, in_texture_coord).rgba;
    #else
        return material.diffuse_factor;
    #endif
}

#ifdef NORMAL_MAP
    in mat3 tbn_matrix;
    in float green_channel_scalar;
#endif

vec3 get_normal() {
    #ifdef NORMAL_MAP
        vec3 material_normal = texture(material.normal_map, in_texture_coord).rgb;
        material_normal = normalize(material_normal * 2.0 - 1.0);
        material_normal = normalize(tbn_matrix * material_normal);
    #else
        vec3 material_normal = in_normal;
    #endif
    return normalize(material_normal);
}

vec4 get_emissive() {
    #ifdef EMISSIVE_MAP
        return texture(material.emissive_map, in_texture_coord).rgba * material.emissive_factor;
    #else
        return material.emissive_factor;
    #endif
}

float get_roughness() {
    #ifdef METALLIC_ROUGHNESS_MAP
        return texture(material.metallic_roughness_map, in_texture_coord).g * material.roughness_factor;
    #else
        return material.roughness_factor;
    #endif
}

float get_metallic() {
    #ifdef METALLIC_ROUGHNESS_MAP
        return texture(material.metallic_roughness_map, in_texture_coord).b * material.metalness_factor;
    #else
        return material.metalness_factor;
    #endif
}

float get_ao() {
    #ifdef AO_MAP
        return texture(material.ao_map, in_texture_coord).r;
    #else
        return 1.0;
    #endif
}
```

Listing 5.16: Sampling the material properties. Defined in *examples/assets/Materials/Shaders/pbr.frag*.

The material properties are based on the glTF 2.0 specification (Khronos Group, 2012b). Each PBR material property has a scalar factor. When the material does not define a texture map for the parameter, the scalar factor defines the material property for the entire primitive.

When the material defines a texture map, the material property is sampled from the texture and scaled against the scalar factor.

In Listing 5.14 line 32, the indirect specular reflections are sampled by using GLSL's `textureLod` function, which allows a shader to perform a texture lookup with an explicit level-of-detail. With the pre-calculated specular light probe, sampling the environment becomes simpler. The reflection vector is used to sample the correct mip-level based on the surface roughness, which gives rough surfaces blurrier reflections. On line 34, the BRDF integration map is then sampled using the viewing angle and the surface roughness as the texture coordinates. This returns the scale and bias to F_0. The results of this sample can then be combined with the prefiltered colour from the specular light probe to reconstruct the approximated integral result. This can then be combined with the diffuse term to achieve the final IBL result. Figure 5.6 depicts the result of using 5.3 and 5.4 as diffuse and specular light probes for image based lighting.

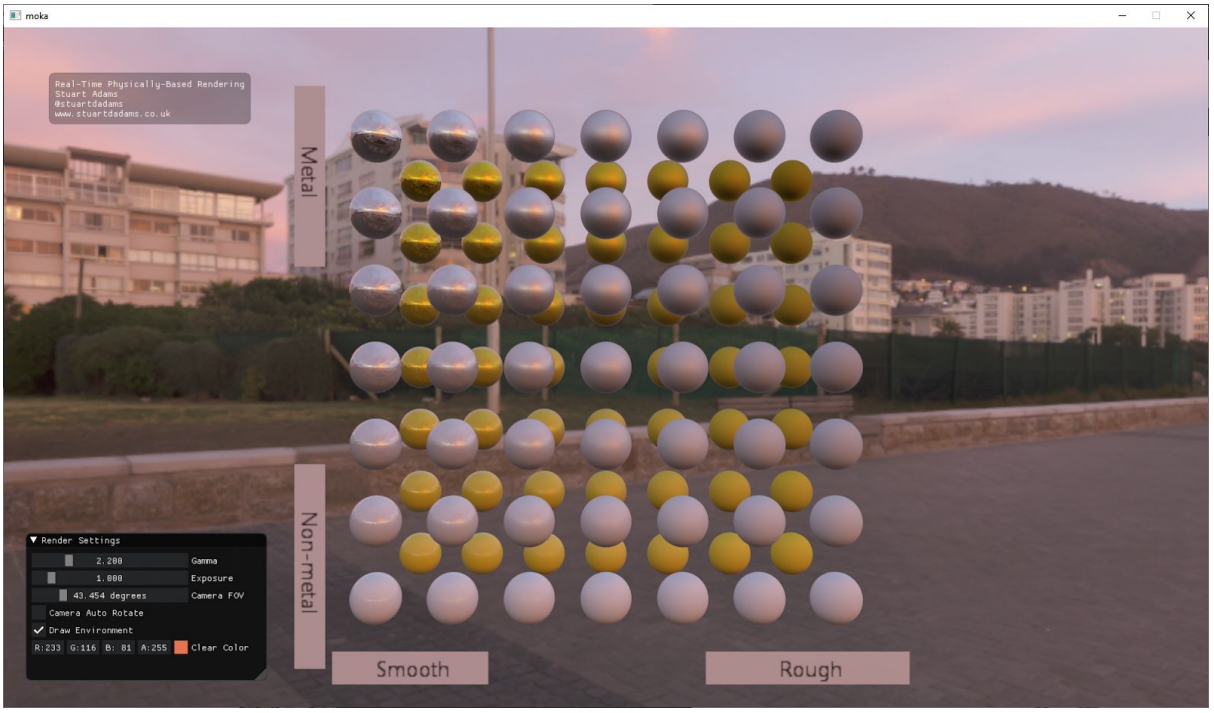


Figure 5.6: A final image rendered by Moka in real-time.

5.2. General Graphics Utilities

5.2.1. glTF Asset Importing

Moka uses the glTF 2.0 model format (Khronos Group (2012a). The test models included in the project were distributed by Khronos Group (2016) to demonstrate the format. A simple glTF loading utility was written that loads an asset using the tinyglTF library and submits the data to the `graphics_device` class. The model loader is basic in design, and pulls the data out of tinyglTF's containers, ensuring that the data is submitted to the graphics device in the format specified by the file. This allows models with different vertex layouts and levels of data precision to be imported. One of the early problems was the flexibility of the materials in

glTF; the format defines many optional texture maps. This means that the same shader cannot necessarily be used for different models, as one may define a different set of texture maps from the other. To ensure that a glTF asset could be imported and shaded, a simple automatic system for creating shader permutations was created.

Moka evaluates the material inputs before the shader is compiled. As materials are processed, a description of the material is built, detailing all the material inputs and their uses. Moka hashes the material description and maintains a lookup table of loaded shaders, using the hashed value as a key. At the end of the primitive importing process, Moka performs a lookup to see if a shader capable of rendering a material of that description exists. If a shader exists, it is used. If a shader does not, it is created by using conditional compilation techniques, including all the code snippets necessary to deal with those material inputs. For GLSL-based shaders, this means using the preprocessor in a C-like fashion; inserting the relevant #define flags into the shader source code before compiling. This compiles all the uniforms and their uses in computing the final colour result. This can be seen in the shader code in 5.1.2.

This functionality is handled by the `material_builder` class, a utility that simplifies constructing a complex material with conditional parameters. The `build` method constructs the final material using the material parameters specified by the programmer. This can be seen in Listing 5.17.

```
material_handle material_builder::build() {
    build_shader_source();

    std::hash<std::string> hash;

    const auto key = hash(vertex_shader_src_ + fragment_shader_src_);

    auto& cache = graphics_device_.get_program_cache();

    program_handle program_handle;

    if (cache.exists(key)) {
        program_handle = cache.get_program(key);
    } else {
        const auto vertex_shader =
            graphics_device_.make_shader(shader_type::vertex, vertex_shader_src_);
        const auto fragment_shader = graphics_device_.make_shader(
            shader_type::fragment, fragment_shader_src_);

        program_handle = graphics_device_.make_program(vertex_shader, fragment_shader);

        cache.add_program(program_handle, key);
    }

    material mat = {
        program_handle, std::move(parameters_), alpha_mode_,
        blend_, culling_, polygon_mode_, depth_test_, scissor_test_};

    return graphics_device_.get_material_cache().add_material(std::move(mat));
}
```

Listing 5.17: Building a material. Defined in *engine/src/graphics/material/material_builder.cpp.*

When building a material, the texture properties that have been specified are used to generate the correct preprocessor definitions to compile into the shader. This can be seen in Listing 5.18. Those preprocessor definitions are then injected into the shader source code, which is hashed to create a unique id for this shader. This ensures that the same shader source can be compiled with different preprocessor definitions and have a unique id. If a material is being built with the same shader configuration as a previous shader, it reuses the existing shader program. This ensures that Moka minimises the number of shader programs that are created when a model is imported many similar materials.

```
void material_builder::build_shader_source() {
    std::string compiler_flags;

    if (alpha_mode_ == alpha_mode::mask) {
        compiler_flags.append("#define MASK_ALPHA\n");
    }

    for (const auto& property : texture_maps_) {
        switch (property) {
        case material_property::diffuse_map:
            compiler_flags.append("#define DIFFUSE_MAP\n");
            break;
        case material_property::emissive_map:
            compiler_flags.append("#define EMISSIVE_MAP\n");
            break;
        case material_property::normal_map:
            compiler_flags.append("#define NORMAL_MAP\n");
            break;
        case material_property::metallic_roughness_map:
            compiler_flags.append("#define METALLIC_ROUGHNESS_MAP\n");
            break;
        case material_property::ao_map:
            compiler_flags.append("#define AO_MAP\n");
            break;
        }
    }

    vertex_shader_src_.insert(0, compiler_flags);
    fragment_shader_src_.insert(0, compiler_flags);
    vertex_shader_src_.insert(0, "#version 330 core\n");
    fragment_shader_src_.insert(0, "#version 330 core\n");
}
```

Listing 5.18: Inserting the correct preprocessor definitions into the shader. Defined in *engine/src/graphics/material/material_builder.cpp.*

5.2.2. Sort-Based Draw Call Bucketing

Moka had trouble with glTF models that had transparency. There was no relationship between the material properties and the order in which meshes were submitted for drawing, therefore transparent objects could occlude opaque objects that should be visible through them. The depth test does not consider the transparency of a fragment, resulting in transparent fragments being written to the depth buffer. Figure 5.7 demonstrates this, where the transparent glass of the googles introduces error; occluding the back of the helmet and the skybox after they fail the depth test. The bright orange color buffer makes clear the extent of the error.

Figure 5.7: Transparent goggles cause noticeable error.

To solve this problem, draw calls must be sorted so they are drawn back to front relative to the distance of the primitive to the camera position. Christer Ericson posted an interesting solution to this problem on his blog, proposing the technique of *sort-based draw call bucketing* (Ericson, 2008). Each draw command is paired with an unsigned key. The key is used to sort draw commands before submitting them to the device. The contents of this key are decided by the application. Moka uses an unsigned 64-bit key, which allows an application to take into consideration many material properties and influencing factors when sorting draw commands. Factors that should be sorted first start at the most significant bit and go down towards the lowest significant bit. Moka sorts draw calls by alpha mode, depth and material id. This can be seen in Listing 5.19. Sorting by material id is useful as it groups together draw calls that use the same shader, limiting the cost of shader state changes in the application. Aras Pranckevičius expands on Ericson's technique to describe how to use the highest bits from a float to sort by depth (Pranckevičius, 2014b). This implementation uses unions for type punning, a technique which is well defined in C99, but undefined in C++. Using this code would not be advisable in a strictly standard-conforming codebase. Figure 5.8 shows the same model as 5.7, but with sort-based draw call bucketing enabled.

```
1  static uint32_t depth_to_bits(const float depth) {
2      // http://aras-p.info/blog/2014/01/16/rough-sorting-by-depth/
3      union {
4          float f = 0.0f;
5          uint32_t i;
6      } data;
7
8      data.f = depth;
9
10     return data.i >> 22;
11 }
12
13 static sort_key generate_sort_key(const float depth, const uint16_t material_id, const alpha_mode alpha) {
14     // http://realtimecollisiondetection.net/blog/?p=86
15     return static_cast<sort_key>(material_id) |
16            static_cast<sort_key>(depth_to_bits(depth)) << 16 |
17            static_cast<sort_key>(alpha == alpha_mode::blend) << 48;
18 }
```

Listing 5.19: Generating a sort key. Defined in *engine/includes/graphics/pbr_scene.hpp.*

Figure 5.8: Transparent goggles are fixed after implementing sort-based draw call bucketing.

With sort-based draw call bucketing implemented, Moka can import and render objects with complex transparent geometry.

5.2.3. OpenGL Error Checking

While third party graphics debugging tools like NSight and RenderDoc can verify the correctness of OpenGL code, there are still situations where runtime debug messages would be useful. When programming without a third party graphics debugger, OpenGL can be difficult to debug. Moka defines useful utilities for catching OpenGL errors that are conditionally compiled into the source. OpenGL errors do not give much information on the API call responsible for triggering the error, so Moka defines this function to spit the last function called to help narrow down the source of the error. This can be seen in Listing 5.20.

```
1  void gl_graphics_api::check_errors(const char* caller) {
2      while (glGetError() != GL_NO_ERROR) {
3          log_.error("OpenGL error discovered. Last call: {}", caller);
4      }
5  }
```

Listing 5.20: Checking , defined in *engine/src/graphics/api/gl_graphics_api.cpp*.

An invocation of the function defined in Listing 5.20 can be seen in Listing 5.21. The calling code will perform all the necessary OpenGL functions before invoking `check_errors` with the name of the function.

```
1  void gl_graphics_api::visit(generate_mipmaps_command& cmd) {
2
3      glBindTexture(GL_TEXTURE_CUBE_MAP, cmd.texture.id);
4      glGenerateMipmap(GL_TEXTURE_CUBE_MAP);
5
6      if constexpr (application_traits::is_debug_build) {
7          check_errors("visit generate_mipmaps_command");
8      }
9  }
```

Listing 5.21: *gl_graphics_api::visit*, defined in *engine/src/graphics/api/gl_graphics_api.cpp*.

C++17's `if constexpr` ensures these functions are not compiled in release builds. This is preferable to slicing up the code with preprocessor use every time conditional compilation is needed. The `application_traits` struct is a simple wrapper around the compile-time constant. This can be seen in Listing 5.22.

```
struct application_traits final {
#ifdef NDEBUG
    constexpr static bool is_debug_build = false;
#else
    constexpr static bool is_debug_build = true;
#endif
};
```

Listing 5.22: Constexpr variables defining the build type. Defined in *engine/includes/application/logger.hpp*.

The second debug output is OpenGL's debug output extension, which can be seen in Listings 5.23 and 5.24. With this extension, it is possible to register a callback to allow the application to print debug messages and to annotate GL objects with human-readable names.

```
if constexpr (application_traits::is_debug_build)
{
    glEnable(GL_DEBUG_OUTPUT);
    glDebugMessageCallback(message_callback, nullptr);
}
```

Listing 5.23: Registering `message_callback` as the debug console output callback. Defined in *engine/src/graphics/api/gl_graphics_api.cpp*.

```
void GLAPIENTRY gl_graphics_api::message_callback(
    const GLenum source,
    const GLenum type,
    const GLuint id,
    const GLenum severity,
    const GLsizei length,
    const GLchar* message,
    const void* user_param) {
    switch (severity) {
    case GL_DEBUG_SEVERITY_HIGH:
        log_.error("message: {}, type: {}, source: {}, id: {}",
            message, type_to_string(type), source_to_string(source), id_to_string(id));
        break;
    case GL_DEBUG_SEVERITY_MEDIUM:
        log_.warn("message: {}, type: {}, source: {}, id: {}",
            message, type_to_string(type), source_to_string(source), id_to_string(id));
        break;
    case GL_DEBUG_SEVERITY_LOW:
        log_.debug("message: {}, type: {}, source: {}, id: {}",
            message, type_to_string(type), source_to_string(source), id_to_string(id));
        break;
    default:
        log_.trace("message: {}, type: {}, source: {}, id: {}",
            message, type_to_string(type), source_to_string(source), id_to_string(id));
        break;
    }
}
```

Listing 5.24: Printing an appropriate log message. Defined in *engine/src/graphics/api/gl_graphics_api.cpp*.

With these debugging facilities in place, it is easy to track down the source of an issue. Figure 5.9 demonstrates the error messages Moka produces when it tries to compile a shader that contains spelling mistakes.

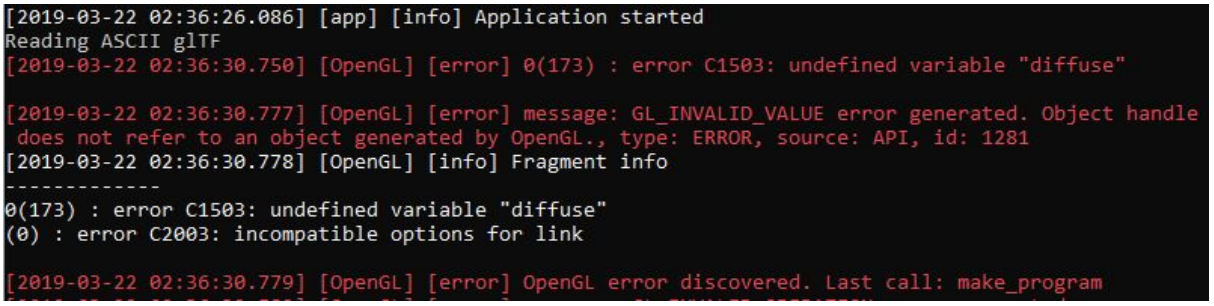


Figure 5.9: Moka console output.

Now the error information is verbose. Not only is the source of error clear, an undefined variable has been referenced in a shader, it also points the programmer to the last call made in the Moka rendering API. This accelerates reproducing and eliminating errors.

6. Results and Evaluation

6.1. PBR Renderer Results

Figure 6.1 shows renders produced by the final Moka implementation. Figure 6.2 and 6.3 show the project running on Windows and Linux desktops.

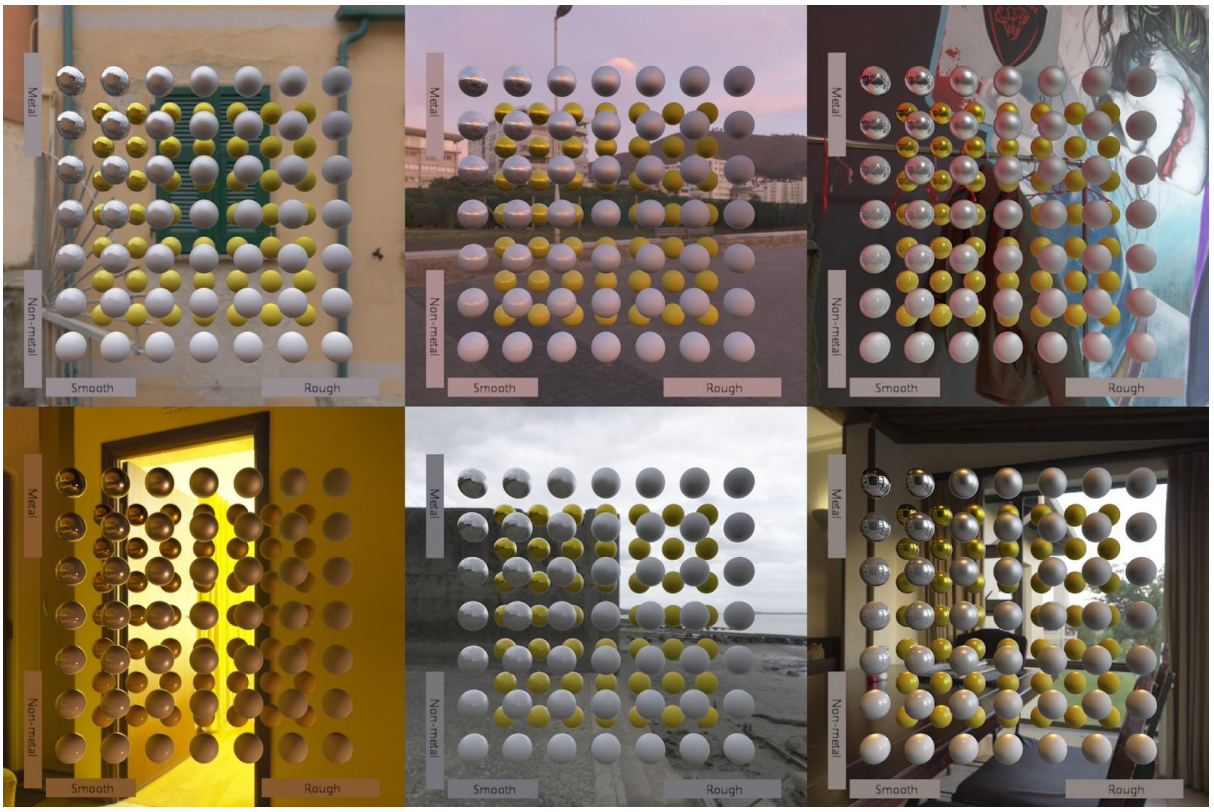


Figure 6.1: Moka showing its effective material system, allowing metals, dielectrics, smooth surfaces and rough surfaces.

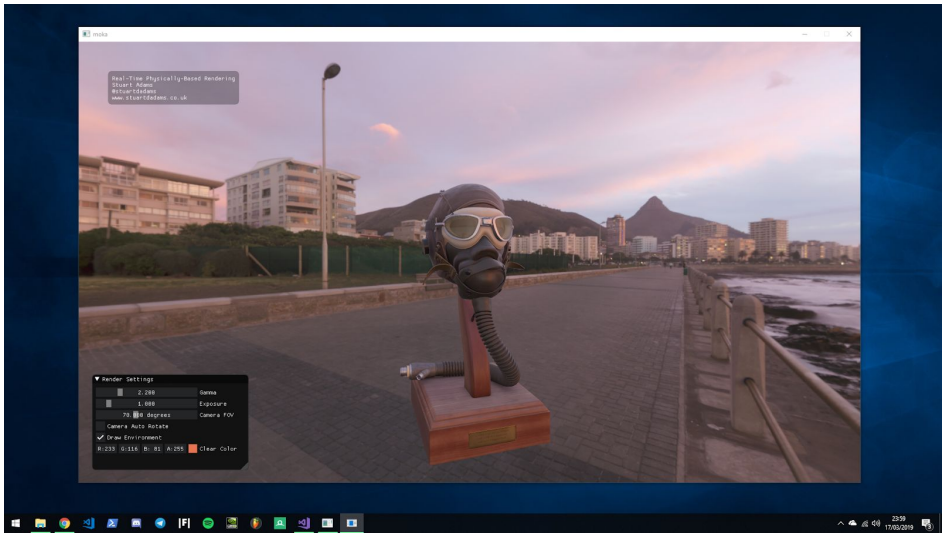


Figure 6.2: x64 Windows 10 Pro, Intel Core i7-4790K, Nvidia GTX 970

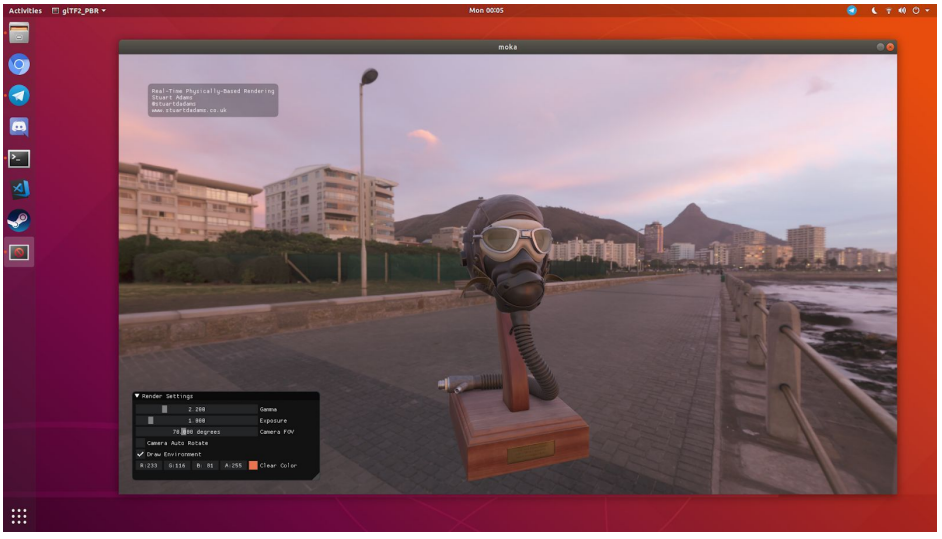


Figure 6.3: x64 Ubuntu 18.04 LTS, Intel Core i7-4790K, Nvidia GTX 970

6.2. Evaluation

Moka renderings were tested against those produced using commercial game engines through an assessment task, presented to participants in the form of a questionnaire. The questionnaire asked participants to determine the ‘realism of the shading’ of 10 images. Respondents were gathered from UWS creative technologies students and members of the public. See Appendix D for the user evaluation questionnaire.

Each participant signed two copies of the consent form. The consent forms can be seen in Appendix E. They were also given a copy of the Plain Language Statement, which detailed the experiment. The Plain Language Statement can be seen in Appendix C.

To test Moka, the survey presents renders from Moka alongside renders from UE4. As discussed in Chapter 1, one of PBR's goals is to create a shading model in which materials are plausible in any lighting environment. To prove that Moka had achieved this goal, the survey tested the effectiveness of the PBR implementation at producing realistic images regardless of the models used or the lighting environment. 10 images were created, featuring a random selection of glTF models and HDR environment maps. 5 of these images were produced using Moka, and the remaining 5 were produced using UE4. The UE4 *Advanced Lighting* project was used. Participants were not informed what engine had produced the images. Figure 6.4 shows two images used in the survey.

Figure 6.4: A comparison between Moka's shading (left) and UE4's shading (right).

The questionnaire features 10 instances of the question "How realistic is the shading of this model?". Respondents are given 5 options: "Very realistic", "Realistic", "Neither Realistic Nor Unrealistic", "Unrealistic", and "Very Unrealistic". Each instance of the question is paired with a different rendering.

The target population for the survey was formed by people that have experience with computer games. PBR as a technology is designed to impress game players, so it is important that the respondents have an interest in computer games and are familiar with the quality of graphics in modern games. There were 30 respondents. 23 of the respondents were members of the UWS *School of Computing, Engineering, and Physical Sciences,* who were approached in the University; and the remaining 7 were members of the public, who were approached in a local gaming cafe.

6.2.1. Evaluation Results

As the questionnaire poses the same question for all 10 images, the responses were pooled together and considered as an experiment with 155 repetitions for each factor, UE4 and Moka. Responses were coded using an integer value between 1 and 5, where 1 represents

"Very Unrealistic" and 5 represents "Very Realistic". Figure 6.5 depicts a Box plot of the response data. It shows that the median responses of both renderers is a 4, "Realistic".

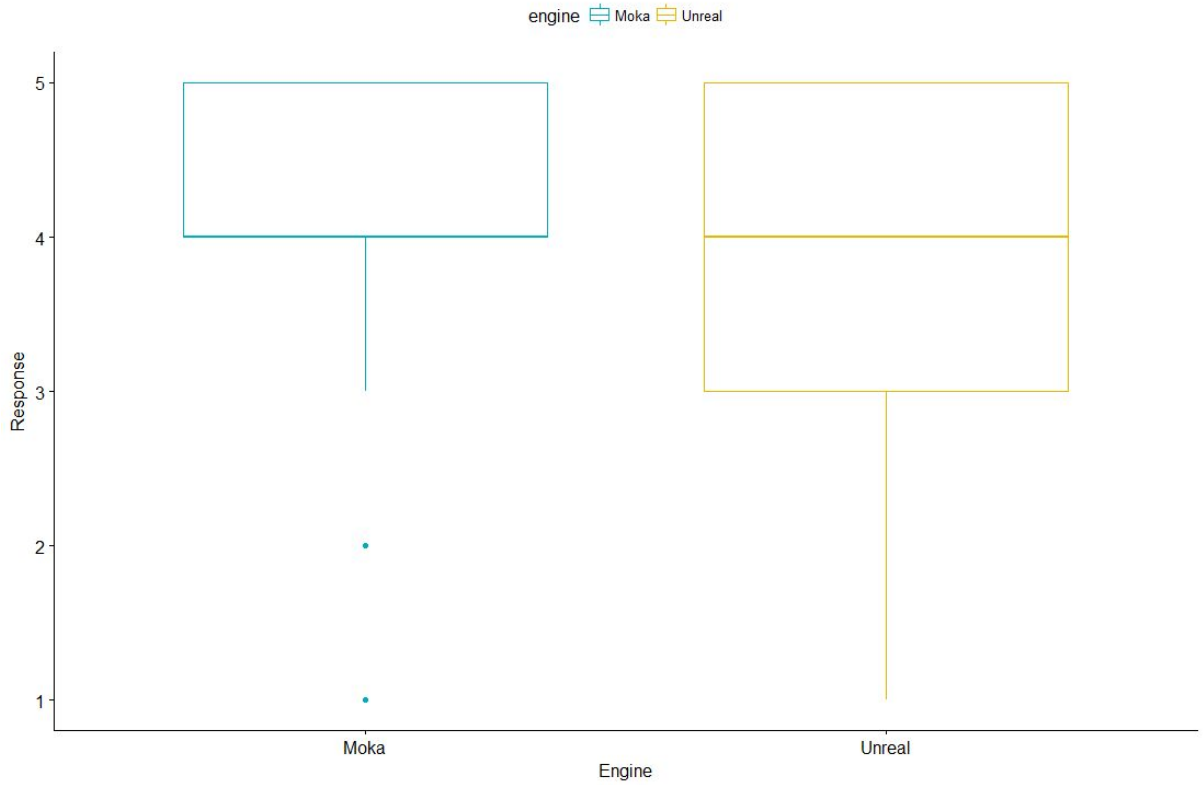


Figure 6.5: A Box Plot of the Response Data

A hypothesis test was performed with a 0.05 significance level on the results due to the high number of repetitions. The null and alternative hypotheses are defined:

H0: *"There is no difference in the realism of shading between UE4 and Moka"*
H1: *"There is difference in the realism of shading between UE4 and Moka"*

The response data is discrete and a normal distribution cannot be assumed. A Mann-Whitney-Wilcoxon non-parametric test was performed using RStudio (see Figure 6.6).

```
> unreal = Analysis$Unreal_resp
> moka = Analysis$Moka_resp
> unrealMedian = median(unreal)
> mokaMedian = median(moka)
> wilcox.test(unreal,moka)

        Wilcoxon rank sum test with continuity correction

data:  unreal and moka
W = 10812, p-value = 0.1039
alternative hypothesis: true location shift is not equal to 0
```

Figure 6.6: Running the test in RStudio.

A p_value of 0.1039 was returned. As the p-value was larger than the significance level, there was no evidence to reject the null hypothesis. It could therefore be concluded that the targeted

population consider Moka's realism of shading to be comparable with that of Unreal Engine. The results of the statistical analysis suggested that when rendering a simple scene, the average game player cannot tell the difference between the Moka implementation of PBR and a full implementation from a commercial game engine. This suggested that Moka's implementation of UE4's techniques as described by Karis (2013) succeeded.

6.2.2. Limitations

The results of the survey cannot suggest that Moka is a match for UE4 in graphical quality. The scenes are set up to showcase UE4's physically-based rendering and image based lighting techniques, giving the questionnaire a direct comparison between UE4's implementation of these techniques and Moka's implementation. They do not show UE4's ability to depict complex, dynamic scenes with physically-based lighting, global illumination, ambient occlusion and screen-space reflections. The research only suggests that Moka's implementation of a small, core subset of UE4's rendering capabilities was successful.

The research depends upon a small sample size of 30 subjects. These subjects were approached face-to-face and asked if they would like to take part in the research. Of the 30 respondents, 23 were affiliated with UWS, and 7 were members of the public. It can be considered sample bias that 76% of the respondents are from the university, as participants affiliated with the University share a higher level of education. Additionally, many of the students were game development students, who have a greater insight into computer graphics than the average consumer. Participants should have been chosen at random while still adhering to the criteria of the study. This would have removed any sample bias that may have affected the outcome of the research.

Two subjects were confused by the questions. They did not understand what "shading" meant in computer graphics, and instead rated the realism of the scene and the subject rather than the realism of the lighting. The questionnaire should have explained the concept of shading in a manner approachable by an average consumer.

The experiment was written to test each renderer ability to take an arbitrary selection of models and lighting environments and produce plausible results. A separate experiment may have been useful to draw direct comparisons between Moka and UE4, using images where the same models are rendered within the same lighting environments.

7. Discussion and Conclusion

7.1. Key Findings

The literature review discussed many complex graphics programming concepts, including; the physics of light-material interaction; microfacet theory; bidirectional reflectance distribution functions; Monte-Carlo sampling; and importance sampling. It was absolutely necessary to have a solid understanding of these subject areas to implement the renderer. It has become clear that implementing a PBR system requires a significant amount of research and a deep understanding of graphics programming techniques.

The user evaluation found that the sample population considered Moka's renderings to be comparable in quality to UE4. Many of the techniques Moka relies on are derived from UE4 itself, so it can be concluded that Moka's implementation of these techniques are successful.

7.2. Suggested Future Work

Many improvements could be made to the Moka project, both in terms of code design and improvements to the rendering.

7.2.1. Improving the Rendering

Moka's rendering could be improved by implementing more advanced graphical techniques. Moka has no support for bloom - an effect that would be well suited to the HDR rendering Moka uses. Bloom results from light scattering in the atmosphere or within the eyes (James and O'Rorke, 2004). Modern graphics hardware can reproduce these effects with simple image processing techniques.

During development, a new paper was published discussing improvements to the image-based lighting technique Moka uses, *A Multiple-Scattering Microfacet Model for Real-Time Image Based Lighting* (Fdez-Agüera, 2019). It proposes extending Karis' (2013) split-sum approximation technique to incorporate multiple scattering.

Lagarde and Zanuttini (2012) show how to use multiple light probes in a scene to achieve accurate local reflections while avoiding lighting seams and parallax issues. Moka can only sample one global light probe, so this would be a big improvement and would allow Moka to render something more complex, such as the Sponza scene, with image based lighting.

Though this would take Moka closer to the state of the art, it would still suffer from problems typical of image-based lighting systems such as light leaking (discussed further in Section 7.3.2) and displaced reflections. Many applications rely on screen-space reflection for pixel-perfect reflection, it brings its own set of problems, as it fails when a reflected object is not visible from the camera's point of view. McGuire et al. (2017) present a set of techniques

that expand upon precomputed image-based lighting techniques to achieve real-time global illumination.

While Moka features image-based lighting, it does not feature dynamic light sources. The field of physically-based lighting is deserving of its own project. Karis (2013) and Lagarde and Rousiers (2014) have detailed discussions on physically-based light sources. One of the most challenging additions would be the addition of physically-based polygonal area lights. Heitz et al., (2016) have proposed a technique for shading arbitrary polygonal-lights in real time.

7.2.1. Improving the C++ Framework

Moka's implementation could be improved in areas other than graphical realism. It lacks an asset importing stage, and as a result it has to process all lighting information at the start of the application. A separate application could be developed that generates the light probes and writes them to an image file with mipmap support. This would allow the Moka application to read the images and hand them to the GPU, rather than running the image convolution every time.

A more efficient model for a command buffer would abandon runtime polymorphism and define each command as a POD struct. When adding the commands to the command buffer, it could then copy the memory into a contiguous buffer. This would however take longer to develop due to the increased boilerplate memory management, and will produce code that is dense and difficult to comprehend.

In future versions of Moka, tinyglTF will be removed. This library is not performant; it makes a lot of wasteful allocations because of poor use of the C++ standard library. It also commits to a heavyweight JSON library as its primary parser. During development, an alternative library gained traction in the graphics community because of its simplicity and efficiency, *cgltf* (Kuhlmann, 2019). Moka will move to using clgtf.

Users of the Moka application should be able to select the light probes at runtime using a graphical interface.

7.3. Software Limitations

7.3.1. Specular Light Probe Noise

At high roughness levels, some high contrast environments can cause noticeable noise once the lower mip levels are sampled. This can be seen in Figure 7.1. Lagarde and Rousiers (2014) acknowledge this and suggest either increasing the number of samples, or trading the noise for correlated bias. They also suggest removing bright light sources from the probe to avoid noise during integration.

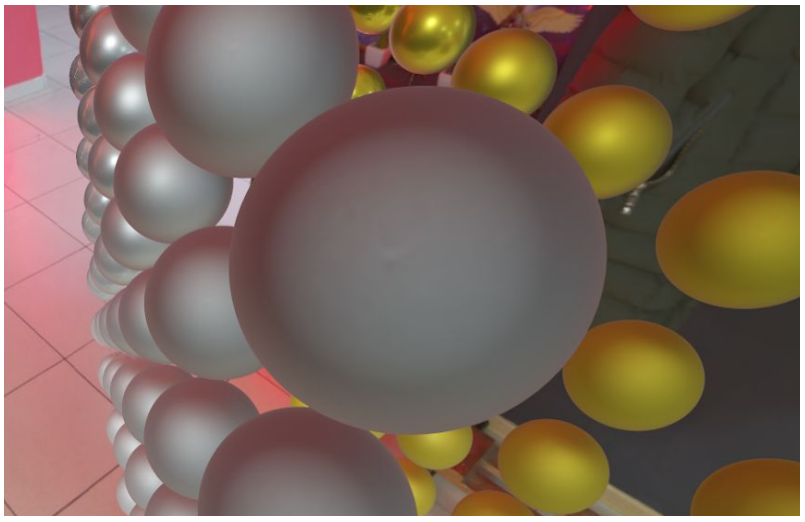


Figure 7.1: Noise at high roughness levels in a high-contrast environment.

7.3.2. Light Leaking

Concave objects suffer from noticeable error as Moka does not prevent shading on the inside faces of models. This can be seen in Figure 7.2. Lagarde and Rousiers (2014) describe this problem as *light leaking*. Although the front of the mask should prevent light from reaching the inside of the mask, it is lit. Lagarde suggests removing strong light sources from the specular probe, and instead use directional lights.

Figure 7.2: Light leaking with concave models in environments with a strong directional light source.

7.3.3. Irradiance Errors

A noticeable source of error can be seen in models with flat faces, such as the base of the flight helmet model. This is demonstrated in Figure 7.3. In this environment, *cave.hdr*, the roof of the cave becomes dark when the irradiance light probe is generated. The base of the model samples this colour upwards. Therefore the base and tip of the mask appear black, while the rest of the model samples the brighter irradiance values. Further research is required to find a solution to this error.

Figure 7.3: Irradiance sampled from the diffuse light probe causes noticeable error in this image. The top of the base is extremely dark, as is the tip of the mask.

7.3.4. Failure to Import High Contrast HDRI Maps

Moka cannot import environment maps with extreme levels of contrast. This can be seen in Figures 7.4 and 7.5. When Moka tries to generate diffuse and specular light probes, the resulting colours become distorted and irregular. Lagarde and Rousiers (2014) mention it is desirable to avoid strong light sources to avoid noise during integration. It is assumed that the sampling algorithm is unsuitable to process HDRI maps that combine very bright areas with very dark areas. This area requires further research to find a solution.

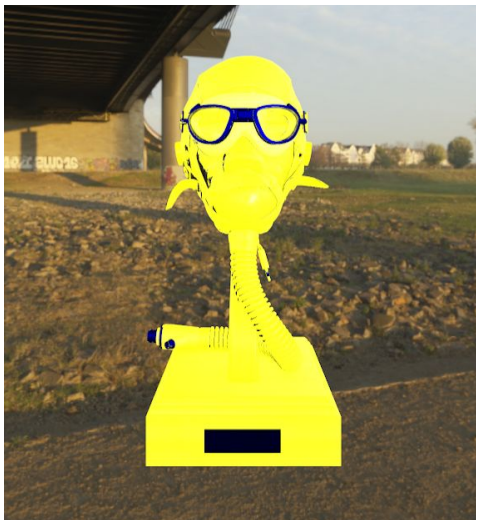


Figure 7.4: Image convolution failure when dealing with HDR maps with extreme contrast levels. This map has extremely bright light sources and large dark areas.

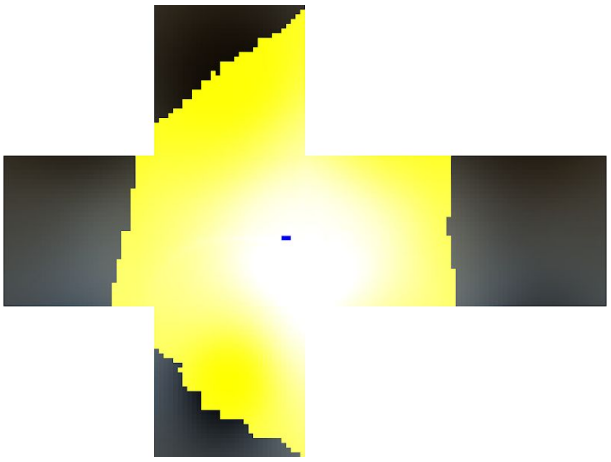


Figure 7.5: The broken specular light probe that was applied to the model in Figure 7.4.

7.4. Conclusions

All of the requirements described in the project specification have been completed to a satisfactory level. The rendering engine produced for this project allows for PBR assets to be imported and rendered using PBR and image-based lighting techniques; the specular microfacet BRDF respects energy conservation and allows fresnel reflections; and Epic Games' split-sum approximation is implemented to achieve real-time specular image-based lighting. Complex glTF assets can be imported and rendered; artists have per-fragment control over material properties such as albedo, metalness, roughness, normals and ambient occlusion; and the material processing will accommodate any valid combination of these properties through its shader permutation system. Renders make use of HDR rendering and gamma correction. In a user evaluation, the target population could not see a substantial difference between the images produced by Moka and images produced by UE4.

The project has a high level of abstraction and the application does not have to interact with the OpenGL rendering API. It is designed for extensibility. The `graphics_device` / `graphics_api` system is designed so the application is agnostic to the native rendering API, and more API backends can be written. Windows and Linux desktops can run the project. The Moka project allows users to produce physically-based images in real-time without an in-depth knowledge of light-material interaction or the native rendering APIs. The project is well documented and users can use Doxygen to compile a comprehensive description of the project's user-facing APIs.

While the implementation succeeded, further development is necessary to bring Moka up to date with the newest developments in image-based lighting techniques. A more in-depth user evaluation study should be conducted that allows users to compare images produced by Moka and competing renderers.

References

Akenine-Möller, T., Haines, E., Hoffman, N., Pesce, A., Iwanicki, M. and Hillaire, S. (2018). *Real-Time Rendering*. 4th ed. Boca Raton, FL: CRC Press.

Baratov, R. (2019). ruslo/hunter. [online] *GitHub*. Available at: https://github.com/ruslo/hunter [Accessed 26 Mar. 2019].

Bjorke, K. (2004). Image-Based Lighting. In: R. Fernando, ed., *GPU Gems*. Boston, MA: Addison-Wesley, pp.307-321.

Burley, B. (2012). Physically-Based Shading at Disney. In: *Practical Physically Based Shading in Film and Game Production*. [online] SIGGRAPH. Available at: https://blog.selfshadow.com/publications/s2012-shading-course/ [Accessed 20 Nov. 2018].

Clang. (2019). ClangFormat. [online] *Clang 9 documentation*. Available at: https://clang.llvm.org/docs/ClangFormat.html [Accessed 29 Mar. 2019].

Cook, R. and Torrance, K. (1982). A Reflectance Model for Computer Graphics. *ACM Transactions on Graphics*, 1(1), pp.7-24.

Cornut, O. (2018). ocornut/imgui. [online] *GitHub*. Available at: https://github.com/ocornut/imgui [Accessed 22 Nov. 2018].

Dammertz, H. (2012). Points on a Hemisphere. [online] *Holger.dammertz.org*. Available at: http://holger.dammertz.org/stuff/notes_HammersleyOnHemisphere.html [Accessed 25 Mar. 2019].

DeVries, J. (2016). *IBL - Diffuse Irradiance*. [online] LearnOpenGL. Available at: https://learnopengl.com/PBR/IBL/Diffuse-irradiance [Accessed 20 Nov. 2018].

DeVries, J. (2016). *IBL - Specular IBL*. [online] LearnOpenGL. Available at: https://learnopengl.com/PBR/IBL/Specular-IBL [Accessed 20 Nov. 2018].

DeVries, J. (2016). *PBR - Theory*. [online] LearnOpenGL. Available at: https://learnopengl.com/PBR/Theory [Accessed 20 Nov. 2018].

Entertainment Software Association (2017). Essential Facts about the Computer and Video Game Industry. [online] *Entertainment Software Association*. Available at: http://www.theesa.com/wp-content/uploads/2017/04/EF2017_FinalDigital.pdf [Accessed 6 Mar. 2018].

Ericson, C. (2008). Order your graphics draw calls around!. [online] *realtimecollisiondetection.net – the blog*. Available at: http://realtimecollisiondetection.net/blog/?p=86 [Accessed 21 Mar. 2019].

Fdez-Agüera, C. (2019). A Multiple-Scattering Microfacet Model for Real-Time Image Based Lighting. *Journal of Computer Graphics Techniques (JCGT)*, [online] 8(1), pp.45-55. Available at: http://jcgt.org/published/0008/01/03/ [Accessed 27 Mar. 2019].

Fujita, S. (2018). syoyo/tinygltf. [online] *GitHub*. Available at: https://github.com/syoyo/tinygltf [Accessed 22 Nov. 2018].

Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. M. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional .

Gotanda, Y. (2012). Beyond a Simple Physically Based Blinn-Phong Model in Real-Time. In: *Practical Physically Based Shading in Film and Game Production*. [online] SIGGRAPH. Available at: https://blog.selfshadow.com/publications/s2012-shading-course/ [Accessed 20 Nov. 2018].

Heitz, E. (2014). Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs. *Journal of Computer Graphics Techniques (JCGT)*, 3(2), pp.48-107.

Heitz, E., Dupuy, J., Hill, S. and Neubelt, D. (2016). Real-Time Polygonal-Light Shading with Linearly Transformed Cosines. *ACM Transactions on Graphics*, [online] 35(4), pp.1-8. Available at: https://dl.acm.org/citation.cfm?doid=2897824.2925895 [Accessed 28 Mar. 2019].

Hoffman, N. (2013). Background: Physics and Math of Shading. In: *Physically Based Shading in Theory and Practice*. [online] SIGGRAPH. Available at: https://blog.selfshadow.com/publications/s2013-shading-course/hoffman/s2013_pbs_physics_math_notes.pdf [Accessed 20 Nov. 2018].

James, G. and O'Rorke, J. (2004). Real-Time Glow. In: R. Fernando, ed., *GPU Gems*. [online] Boston, MA: Addison Wesley. Available at: https://developer.nvidia.com/gpugems/GPUGems/gpugems_ch21.html [Accessed 27 Mar. 2019].

Karadžić, B. (2012). bkaradzic/bgfx. [online] *GitHub*. Available at: https://github.com/bkaradzic/bgfx [Accessed 27 Mar. 2019].

Karis, B. (2013). Real Shading in Unreal Engine 4. In: *Physically Based Shading in Theory and Practice*. [online] SIGGRAPH. Available at: http://gamedevs.org/uploads/real-shading-in-unreal-engine-4.pdf/ [Accessed 20 Nov. 2018].

Khronos Group (2012a). KhronosGroup/glTF. [online] *GitHub*. Available at: https://github.com/KhronosGroup/glTF/ [Accessed 25 Mar. 2019].

Khronos Group (2012b). KhronosGroup/glTF. [online] *GitHub*. Available at: https://github.com/KhronosGroup/glTF/blob/master/specification/2.0/README.md#materials [Accessed 25 Mar. 2019].

Khronos Group (2016). KhronosGroup/glTF-Sample-Models. [online] *GitHub*. Available at: https://github.com/KhronosGroup/glTF-Sample-Models [Accessed 25 Mar. 2019].

Křivánek, J. and Colbert, M. (2007). GPU-Based Importance Sampling. In: H. Nguyen, ed., *GPU Gems 3*. [online] Boston, MA: Addison-Wesley, pp.459 - 477. Available at: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_pref01.html [Accessed 27 Mar. 2019].

Kuhlmann, J. (2019). jkuhlmann/cgltf. [online] *GitHub*. Available at: https://github.com/jkuhlmann/cgltf [Accessed 25 Mar. 2019].

Lagarde, S. (2012). *AMD Cubemapgen for physically based rendering*. [online] Sébastien Lagarde. Available at: https://seblagarde.wordpress.com/2012/06/10/amd-cubemapgen-for-physically-based-rendering/ [Accessed 20 Nov. 2018].

Lagarde, S. and Rousiers, C. (2014). Moving Frostbite to Physically Based Rendering 3.0. In: *Physically Based Shading in Theory and Practice*. [online] SIGGRAPH. Available at: https://seblagarde.wordpress.com/2015/07/14/siggraph-2014-moving-frostbite-to-physically-based-rendering/ [Accessed 20 Nov. 2018].

Lagarde, S. and Zanuttini, A. (2012). Local image-based lighting with parallax-corrected cubemaps. *ACM SIGGRAPH 2012*.

Lantinga, S. (2018). Simple DirectMedia Layer. [online] *Libsdl.org*. Available at: https://www.libsdl.org/ [Accessed 22 Nov. 2018].

Lazarov, D. (2013). Getting More Physical in Call of Duty: Black Ops II. In: *Physically Based Shading in Theory and Practice*. [online] SIGGRAPH. Available at: https://blog.selfshadow.com/publications/s2013-shading-course/ [Accessed 20 Nov. 2018].

Lengyel, E. (2012). *Mathematics for 3D Game Programming and Computer Graphics*. 3rd ed. Boston, MS: Course Technology.

McAllister, D. (2004). Spatial BRDFs. In: R. Fernando, ed., *GPU Gems*. Boston, MA: Addison-Wesley, pp.293-306.

McGuire, M., Mara, M., Nowrouzezahrai, D. and Luebke, D. (2017). Real-time global illumination using precomputed light field probes. *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D '17*.

Melman, G. (2018). gabime/spdlog. [online] *GitHub*. Available at: https://github.com/gabime/spdlog [Accessed 22 Nov. 2018].

Meyers, S. (2005). *Effective C++*. 3rd ed. Reading, Massachusetts: Addison-Wesley.

Meyers, S. (2014). *Effective Modern C++*. Beijing: O'Reilly Media.

MonoGame (2019). Documentation. [online] *MonoGame*. Available at: http://www.monogame.net/documentation/?page=T_Microsoft_Xna_Framework_Game [Accessed 23 Mar. 2019].

Oren, M. and Nayar, S. (1994). Generalization of Lambert's reflectance model. *Proceedings of the 21st annual conference on Computer graphics and interactive techniques - SIGGRAPH '94*, pp.239-246.

Pharr, M., Jakob, W. and Humphreys, G. (2017). *Physically Based Rendering*. 3rd ed. Cambridge, MA: Morgan Kaufmann.

Phong, B. (1975). Illumination for computer generated pictures. *Communications of the ACM*, 18(6), pp.311-317.

Pranckevičius, A. (2014). Physically Based Shading in Unity. [online] *GDC*. Available at: https://aras-p.info/texts/files/201403-GDC_UnityPhysicallyBasedShading_notes.pdf [Accessed 6 Mar. 2018].

Pranckevičius, A. (2014). Rough sorting by depth · *Aras' website*. [online] Aras' website. Available at: http://aras-p.info/blog/2014/01/16/rough-sorting-by-depth/ [Accessed 22 Mar. 2019].

Pranckevičius, A. (2015). Unity 5 Graphics Smörgåsbord. [online] *GDC*. Available at: https://aras-p.info/texts/files/201503-GDC_Unity5_Graphics_notes.pdf [Accessed 6 Mar. 2018].

Ramamoorthi, R. and Hanrahan, P. (2001). An efficient representation for irradiance environment maps. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01*, pp.497-500.

Riccio, C. (2018). OpenGL Mathematics. [online] *Glm.g-truc.net*. Available at: https://glm.g-truc.net/0.9.9/index.html [Accessed 22 Nov. 2018].

Schlick, C. (1994). An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum*, 13(3), pp.233-246.

Schwaber, K. and Sutherland, J. (2016). The Scrum Guide. [online] *Scrumguides.org*. Available at:

https://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf [Accessed 29 Mar. 2019].

Smith, B. (1967). Geometrical shadowing of a random rough surface. *IEEE Transactions on Antennas and Propagation*, 15(5), pp.668-671.

Spogreev, I. (2016). Physically Based Light Probe Generation on GPU. In: W. Engel, ed., *GPU Pro 6*. Boca Raton, FL: CRC Press, pp.243-266.

Torrance, K. and Sparrow, E. (1967). Theory for Off-Specular Reflection From Roughened Surfaces. *Journal of the Optical Society of America*, 57(9), p.1105.

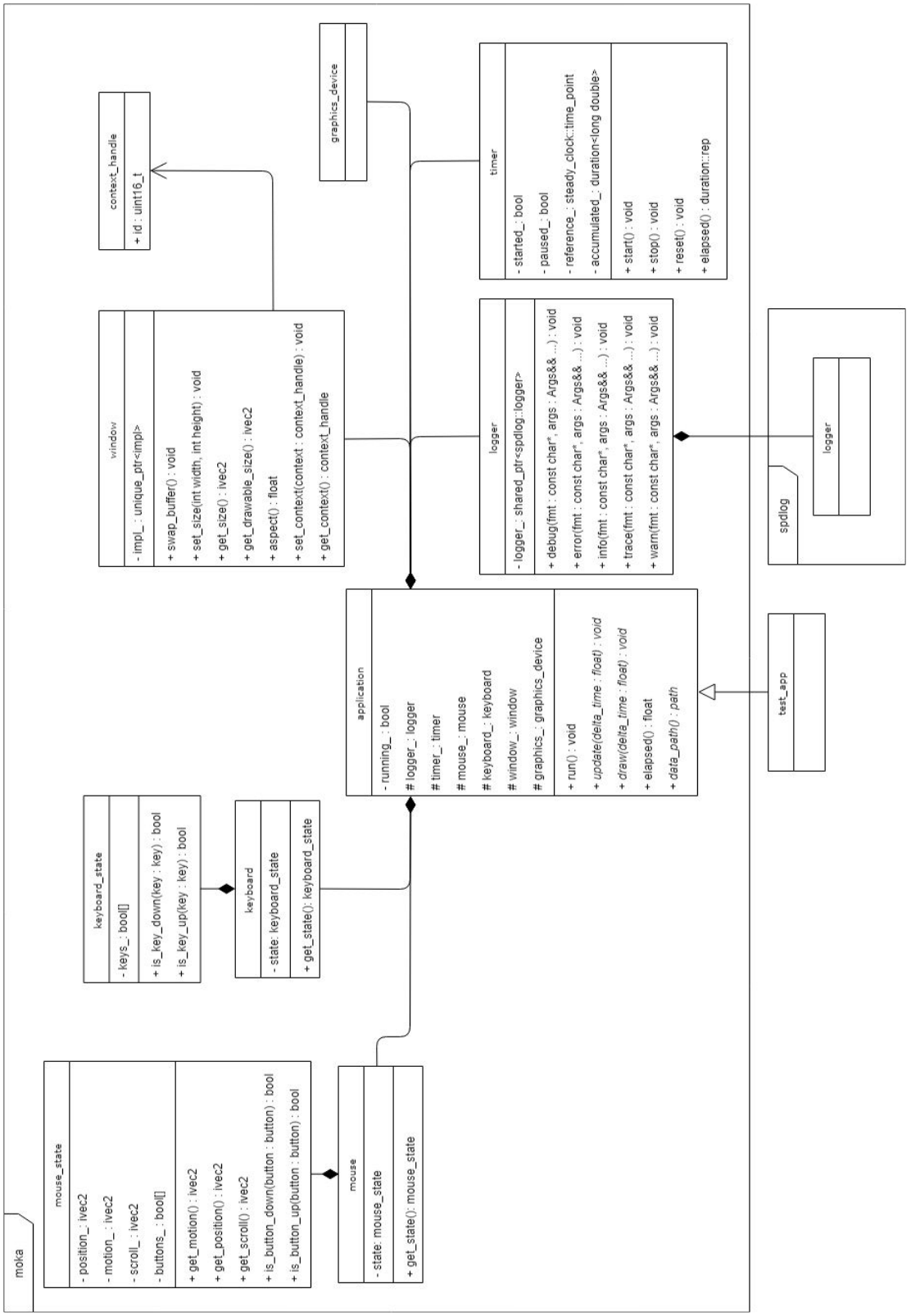
Trowbridge, T. and Reitz, K. (1975). Average irregularity representation of a rough surface for ray reflection. *Journal of the Optical Society of America*, 65(5), pp.531-536.

Walter, B., Marschner, S., Li, H. and Torrance, K. (2007). Microfacet models for refraction through rough surfaces. *Proceedings of the 18th Eurographics conference on Rendering Techniques - EGSR '07*, pp.195-206.

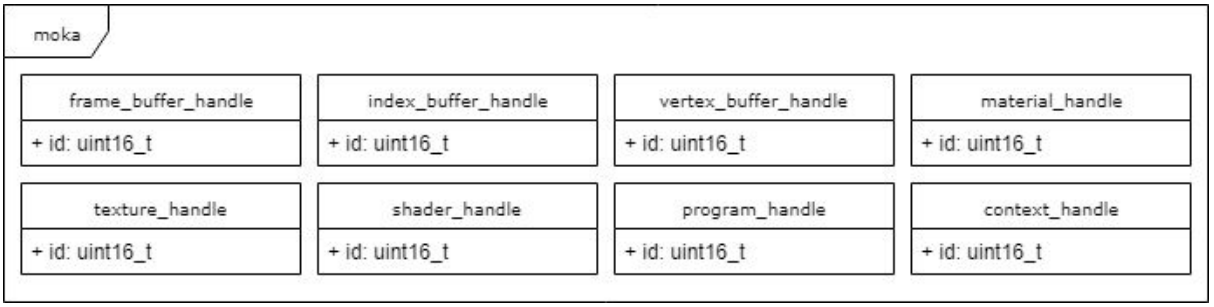
Ward, G. (1991). Real Pixels. In: J. Arvo, ed., *Graphics Gems II*. Ithaca, NY: Academic Press, pp.80-83.

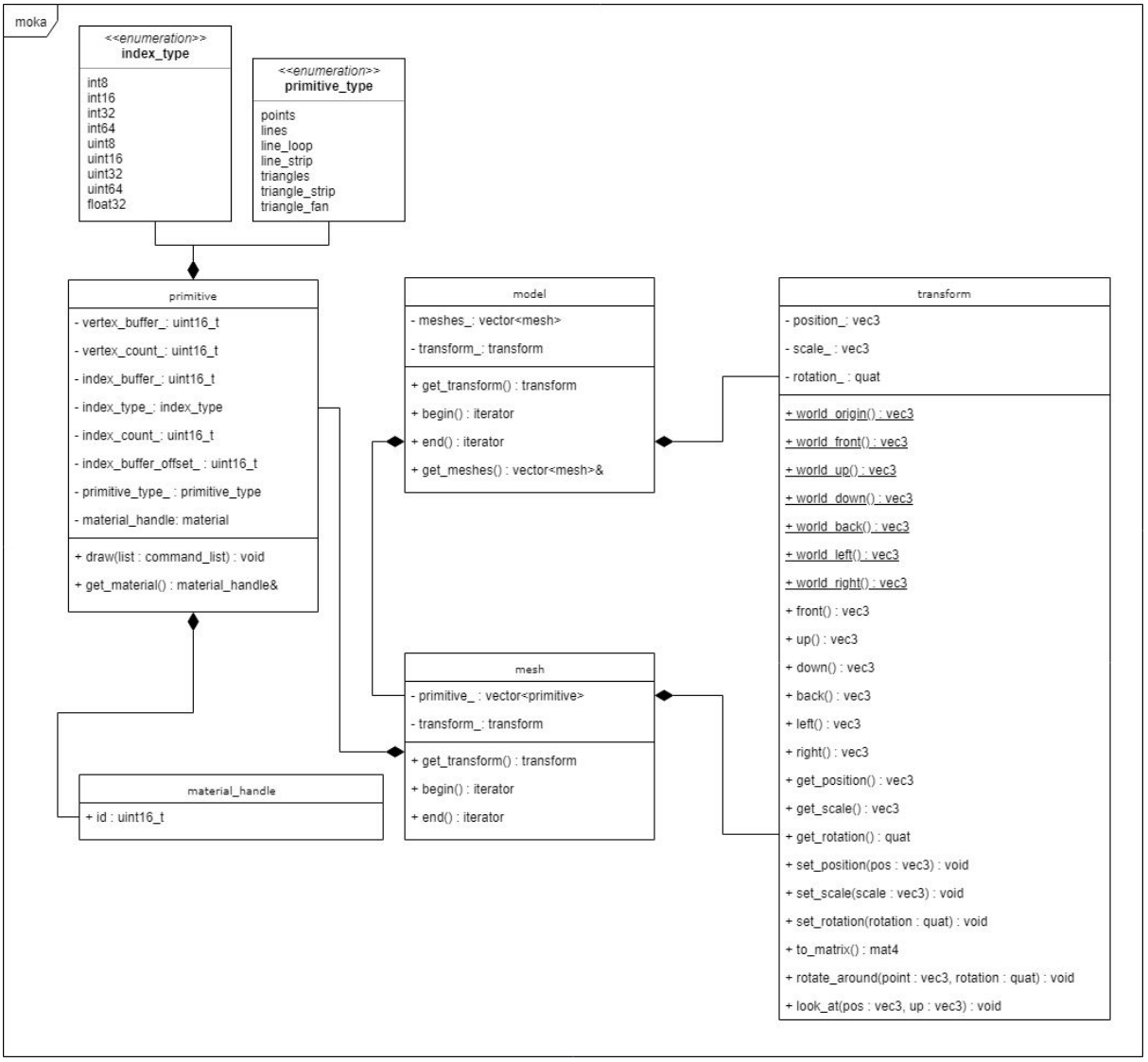
Zaal, G. (2019). HDRI Haven. [online] *HDRI Haven*. Available at: https://hdrihaven.com/ [Accessed 25 Mar. 2019].

Appendix A: UML Diagrams




moka
<<Interface>>
graphics_api
+ visit(cmd : clear_command) : void
+ visit(cmd : draw_command) : void
+ visit(cmd : viewport_command) : void
+ visit(cmd : scissor_command) : void
+ visit(cmd : fill_vertex_buffer_command) : void
+ visit(cmd : fill_index_buffer_command) : void
+ visit(cmd : frame_buffer_command) : void
+ visit(cmd : frame_buffer_texture_command) : void
+ visit(cmd : generate_mipmaps_command) : void
+ visit(cmd : set_material_parameters_command) : void
+ submit(commands : command_list) : void
+ submit_and_swap(commands : command_list) : void
+ make_program(vertex_handle : render_texture_data, render_texture_count : size_t) : frame_buffer_handle
+ make_frame_buffer(render_textures: shader_type, source : string) : shader_handle
+ make_vertex_buffer(data : void*, size : size_t, layout : vertex_layout, use : buffer_usage) : vertex_buffer_handle
+ make_index_buffer(data : void*, size : size_t, type : index_type, use : buffer_usage) : index_buffer_handle
+ make_texture(data : void*, resolution : ivec2, components : texture_components, wrap_mode : texture_wrap_mode, has_mipmaps : bool) : texture_handle
gl_graphics_api
+ visit(cmd : clear_command) : void
+ visit(cmd : draw_command) : void
+ visit(cmd : viewport_command) : void
+ visit(cmd : scissor_command) : void
+ visit(cmd : fill_vertex_buffer_command) : void
+ visit(cmd : fill_index_buffer_command) : void
+ visit(cmd : frame_buffer_command) : void
+ visit(cmd : frame_buffer_texture_command) : void
+ visit(cmd : generate_mipmaps_command) : void
+ visit(cmd : set_material_parameters_command) : void
+ submit(commands : command_list) : void
+ submit_and_swap(commands : command_list) : void
+ make_program(vertex_handle : render_texture_data, render_texture_count : size_t) : frame_buffer_handle
+ make_frame_buffer(render_textures: shader_type, source : string) : shader_handle
+ make_vertex_buffer(data : void*, size : size_t, layout : vertex_layout, use : buffer_usage) : vertex_buffer_handle
+ make_index_buffer(data : void*, size : size_t, type : index_type, use : buffer_usage) : index_buffer_handle
+ make_texture(data : void*, resolution : ivec2, components : texture_components, wrap_mode : texture_wrap_mode, has_mipmaps : bool) : texture_handle
graphics_device
- graphics_api_ : std::unique_ptr<graphics_api>
- thread_ : thread
- mutex_ : mutex
+ submit(commands : command_list) : void
+ submit_and_swap(commands : command_list) : void
+ make_program(vertex_handle : render_texture_data, render_texture_count : size_t) : frame_buffer_handle
+ make_frame_buffer(render_textures: shader_type, source : string) : shader_handle
+ make_vertex_buffer(data : void*, size : size_t, layout : vertex_layout, use : buffer_usage) : vertex_buffer_handle
+ make_index_buffer(data : void*, size : size_t, type : index_type, use : buffer_usage) : index_buffer_handle
+ make_texture(data : void*, resolution : ivec2, components : texture_components, wrap_mode : texture_wrap_mode, has_mipmaps : bool) : texture_handle
+ build_material() : material_builder
+ build_texture() : texture_builder
+ build_frame_buffer() : frame_buffer_builder
command_list
- command_buffers_ : vector<command_buffer>
- is_sorted_ : bool
+ accept(graphics_device :graphics_device&) : void
+ clear() : clear_command&
+ draw() : draw_command&
+ scissor() : scissor_command&
+ viewport() : viewport_command&
+ fill_index_buffer() : fill_index_buffer_command&
+ fill_vertex_buffer() : fill_vertex_buffer_command&
+ generate_mipmaps() : generate_mipmaps_command&
+ set_material_parameters() : set_material_parameters_command&
+ frame_buffer() : frame_buffer_command&
+ frame_buffer_texture() : frame_buffer_texture_command&
command_buffer
- commands_ : vector<unique_ptr<graphics_command>>
- id_ : sort_key
+ accept(graphics_device : graphics_device&) : void
+ clear() : clear_command&
+ draw() : draw_command&
+ scissor() : scissor_command&
+ viewport() : viewport_command&
+ fill_index_buffer() : fill_index_buffer_command&
+ fill_vertex_buffer() : fill_vertex_buffer_command&
+ generate_mipmaps() : generate_mipmaps_command&
+ set_material_parameters() : set_material_parameters_command&
+ frame_buffer() : frame_buffer_command&
+ frame_buffer_texture() : frame_buffer_texture_command&
<<Interface>>
graphics_command
+ accept(api : graphics_api) : void
viewport_command
+ accept(api : graphics_api) : void
draw_command
+ accept(api : graphics_api) : void
set_material_parameters_command
+ accept(api : graphics_api) : void
clear_command
+ accept(api : graphics_api) : void
frame_buffer_command
+ accept(api : graphics_api) : void
scissor_command
+ accept(api : graphics_api) : void
frame_buffer_texture_command
+ accept(api : graphics_api) : void
fill_index_buffer_command
+ accept(api : graphics_api) : void
fill_vertex_buffer_command
+ accept(api : graphics_api) : void


moka
frame_buffer_handle
+ id: uint16_t
index_buffer_handle
+ id: uint16_t
vertex_buffer_handle
+ id: uint16_t
material_handle
+ id: uint16_t
texture_handle
+ id: uint16_t
shader_handle
+ id: uint16_t
program_handle
+ id: uint16_t
context_handle
+ id: uint16_t


moka
<<enumeration>>
index_type
int8
int16
int32
int64
uint8
uint16
uint32
uint64
float32
<<enumeration>>
primitive_type
points
lines
line_loop
line_strip
triangles
triangle_strip
triangle_fan
primitive
- vertex_buffer_: uint16_t
- vertex_count_: uint16_t
- index_buffer_: uint16_t
- index_type_: index_type
- index_count_: uint16_t
- index_buffer_offset_ : uint16_t
- primitive_type_ : primitive_type
- material_handle: material
+ draw(list : command_list) : void
+ get_material() : material_handle&
model
- meshes_: vector<mesh>
- transform_: transform
+ get_transform() : transform
+ begin() : iterator
+ end() : iterator
+ get_meshes() : vector<mesh>&
transform
- position_: vec3
- scale_ : vec3
- rotation_ : quat
+ world_origin() : vec3
+ world_front() : vec3
+ world_up() : vec3
+ world_down() : vec3
+ world_back() : vec3
+ world_left() : vec3
+ world_right() : vec3
+ front() : vec3
+ up() : vec3
+ down() : vec3
+ back() : vec3
+ left() : vec3
+ right() : vec3
+ get_position() : vec3
+ get_scale() : vec3
+ get_rotation() : quat
+ set_position(pos : vec3) : void
+ set_scale(scale : vec3) : void
+ set_rotation(rotation : quat) : void
+ to_matrix() : mat4
+ rotate_around(point : vec3, rotation : quat) : void
+ look_at(pos : vec3, up : vec3) : void
mesh
- primitive_ : vector<primitive>
- transform_: transform
+ get_transform() : transform
+ begin() : iterator
+ end() : iterator
material_handle
+ id : uint16_t

Appendix B: Critical Self-Appraisal

Throughout this project, I have gained an insight into modern graphics programming problems and solutions. I created this project specification to challenge myself; I chose a mathematics focused project would force me to improve my understanding of linear algebra and introduce me to more complex subject areas. I believe this was the right decision, and I am excited to continue improving my maths skills.

In hindsight, the project was far too broad. A specific technique should have been chosen to explore and implement. By giving myself the umbrella of PBR to explore, I resigned myself to explaining and exploring a massive field of research. This project pushed the limits of my research, project management, and programming experience. Despite this, I met the time constraints and the software / documentation met the project specification requirements.

I now believe the requirement I set to create a reusable C++ framework to have been a mistake. Moka's development took time and effort away from the PBR subject, and did not contribute to the results of the research. In hindsight, An existing framework like BGFX should have been used, allowing me to focus all of my efforts on creating a good PBR solution.

The application requires more robust error handling when it loads an invalid asset. It should give more useful console output. I wanted the application to present a graphical interface that would allow the user to select the rendering environment and see the model's lighting update, but time limitations forced me to reconsider.

Project management was successful, with some notable exceptions. Meetings with my supervisor were regular and well documented. I created a project schedule, which I referred to and updated throughout the project. I adjusted the project's priorities to ensure that I would submit a project that met all of the requirements agreed upon. While I completed all my goals on time, I gave too much time to the code and too little time to research. The research could have pulled upon a much larger sample size if I had distributed the survey online, but the project did not receive the correct ethical approval to accept online responses.

Moka implements UE4's techniques and convinced the target population of the survey of its ability to create realistic images; I believe this project has accomplished its goals well. I am satisfied with the results of this project. The experience has given me insight into the necessity of proper project planning and time management, which I will carry forward into my professional career.

Appendix C: Plain Language Statement

`

UWS UNIVERSITY OF THE WEST of SCOTLAND

Plain Language Statement Information Sheet

School: School of Computing, Engineering and Physical Sciences

Project Title: Real-Time Physically-Based Rendering

Computing Hons Project Student: Stuart Adams
Email address: B00265262@studentmail.uws.ac.uk

Computing Hons Project Supervisor: Dr. Marco Gilardi
Email address: Marco.Gilardi@uws.ac.uk
Contact number: 0141 846 4379

Computing Hons Project Module Coordinator: Dr Mark Stansfield
Email address: mark.stansfield@uws.ac.uk
Contact number: 0141 848 3963

Programme Title: BSc (Hons) in: *Computer Games Technology*

Dear participant,

You are being invited to take part in my research study – as above. Before you decide it is important for you to understand why the research is being done and what will be involved. Please take time to read the following information carefully and discuss it with others if you wish. Ask us if there is anything that is not clear or if you would like more information. Please take your time in deciding if you wish to take part and thank you for reading this.

What is the purpose of the study?

This research seeks to examine and exemplify the techniques used by modern game engines to achieve photorealism. Physically Based Rendering (PBR) is the practice of simulating the physics of light and matter interaction to achieve photorealistic shading. The objective of this research is to develop a real-time renderer that exemplifies these techniques. The data collected during this experiment will be analyzed to discover how effective the final result is compared to commercial game engines, such as Unity and Unreal. Results obtained using the data in this

experiment will be used to write an academic paper about the effectiveness of PBR. Only the statistics obtained from the data will be used for publication.

Why have I been chosen?

You have been selected as a subject with an appreciation of video games. The technology developed in this research is designed to be appreciated by game players.

Do I have to take part?

It is up to you to decide whether or not to take part. If you do decide to take part you will be given this information sheet to keep and be asked to sign a consent form. If you decide to take part you are still free to withdraw at any time and without giving a reason. A decision not to participate will not affect your grades in any way.

What will happen to me if I take part?

As a participating subject you will be asked to rate a series of 10 computer generated images for their realism.

Will my taking part in this study be kept confidential?

All information, which is collected, about you during the course of the research will be kept strictly confidential. You will be identified by an ID number or letter and any information about you will have your name, address and all other identifiable details removed so that you cannot be recognised from it.

What will happen to the results of the research study?

A bound copy of the completed Hons Project report may be stored at the University of the West of Scotland library (subject to approval).

Who is organising the research?

The School of Engineering and Computing at the University of the West of Scotland is organising this Computing Hons Project.

Who has reviewed the study?

The project has been reviewed by the student's supervisor, moderator, year leader, module coordinator and chair of the School of Engineering and Computing Ethics Committee.

Contact for Further Information

For further information please contact:

Computing Hons Project Student: Stuart Adams

Email address: B00265262@studentmail.uws.ac.uk

Alternatively, if participants have any concerns regarding the conduct of the research project please contact the School of Computing, Engineering and Physical Sciences Ethics Committee Chair by contacting Dr John Hughes – john.hughes@uws.ac.uk.

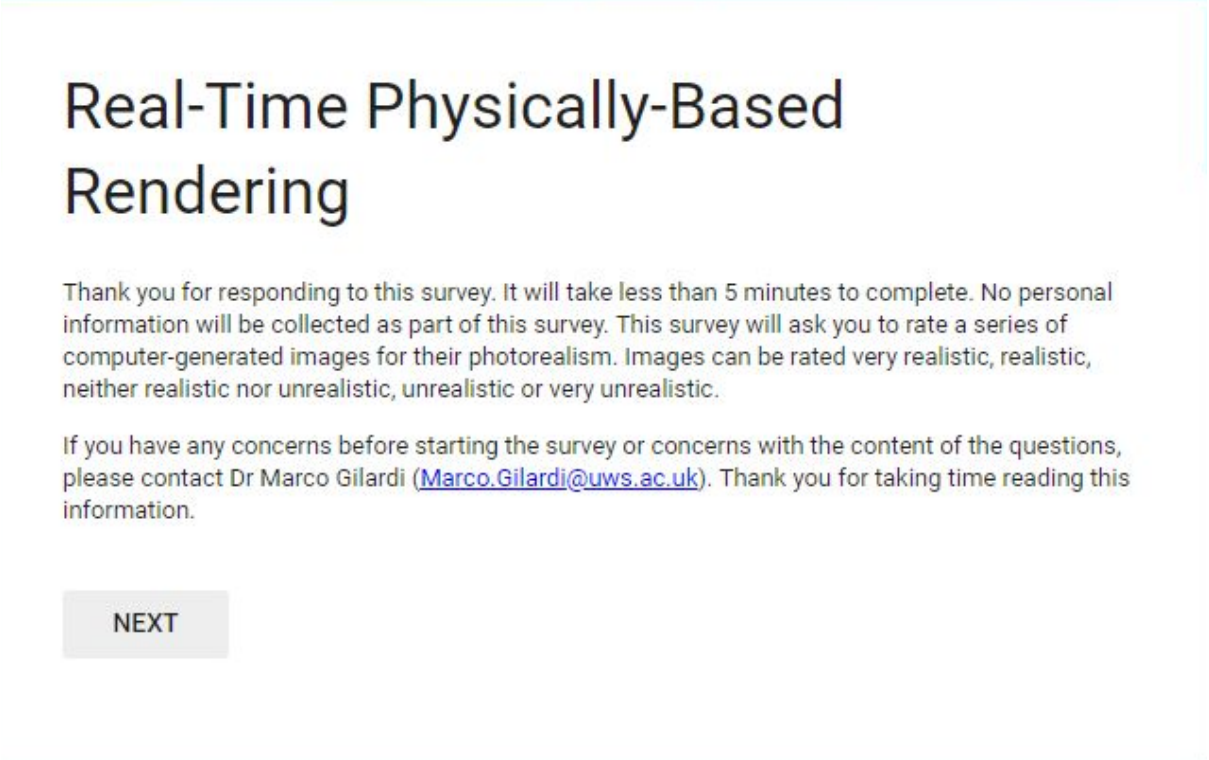
Thank you for taking part in this study.

If you have any queries or concerns relating to the research being undertaken, please contact:

Dr Mark Stansfield
School of Computing
University of the West of Scotland
High Street
Paisley
PA1 2BE

E-mail: mark.stansfield@uws.ac.uk

Appendix D: Survey



Real-Time Physically-Based Rendering

Thank you for responding to this survey. It will take less than 5 minutes to complete. No personal information will be collected as part of this survey. This survey will ask you to rate a series of computer-generated images for their photorealism. Images can be rated very realistic, realistic, neither realistic nor unrealistic, unrealistic or very unrealistic.

If you have any concerns before starting the survey or concerns with the content of the questions, please contact Dr Marco Gilardi (Marco.Gilardi@uws.ac.uk). Thank you for taking time reading this information.

NEXT

Real-Time Physically-Based Rendering

*Required

Computer Generated Images

Please rate the following images for their realism.

How realistic is the shading of this model? *

- ◯ Very realistic
- ◯ Realistic
- ◯ Neither realistic nor unrealistic
- ◯ Unrealistic
- ◯ Very unrealistic

How realistic is the shading of this model? *

- ◯ Very realistic
- ◯ Realistic
- ◯ Neither realistic nor unrealistic
- ◯ Unrealistic
- ◯ Very unrealistic

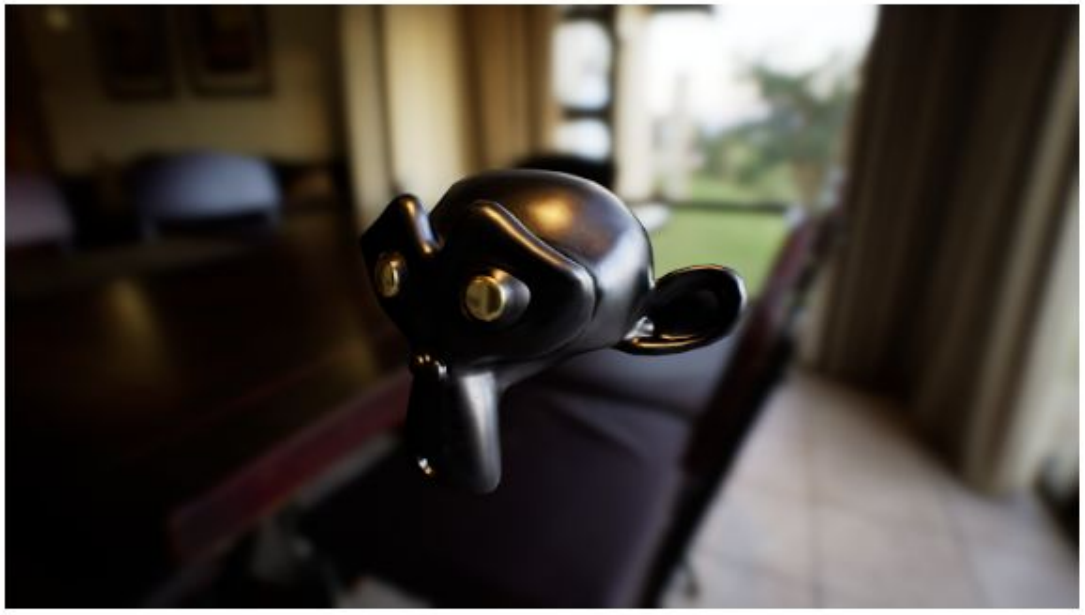
How realistic is the shading of this model? *

- ○ Very realistic
- ○ Realistic
- ○ Neither realistic nor unrealistic
- ○ Unrealistic
- ○ Very unrealistic

How realistic is the shading of this model? *

- ○ Very realistic
- ○ Realistic
- ○ Neither realistic nor unrealistic
- ○ Unrealistic
- ○ Very unrealistic

How realistic is the shading of this model? *



- ○ Very realistic
- ○ Realistic
- ○ Neither realistic nor unrealistic
- ○ Unrealistic
- ○ Very unrealistic

How realistic is the shading of this model? *

- ○ Very realistic
- ○ Realistic
- ○ Neither realistic nor unrealistic
- ○ Unrealistic
- ○ Very unrealistic

How realistic is the shading of this model? *

- ○ Very realistic
- ○ Realistic
- ○ Neither realistic nor unrealistic
- ○ Unrealistic
- ○ Very unrealistic

How realistic is the shading of this model? *

- ○ Very realistic
- ○ Realistic
- ○ Neither realistic nor unrealistic
- ○ Unrealistic
- ○ Very unrealistic

How realistic is the shading of this model? *

- ○ Very realistic
- ○ Realistic
- ○ Neither realistic nor unrealistic
- ○ Unrealistic
- ○ Very unrealistic

How realistic is the shading of this model? *

- ○ Very realistic
- ○ Realistic
- ○ Neither realistic nor unrealistic
- ○ Unrealistic
- ○ Very unrealistic