UNIVERSITY OF THE WEST of SCOTLAND

UWS

BSc (Hons) Computer Games Technology

Real-Time Physically-Based Rendering

Computing Honours Project (COMP10034) Interim Report

Stuart D. Adams
B00265262

29/10/2018

Supervisor: Marco Gilardi

Abstract

This paper discusses techniques for achieving photorealism in real-time applications. The report explores the foundations of the physics of light-matter interaction before discussing how these observations can be applied in the shading process. The initial design and implementation of a real-time renderer is presented, which is being developed to exemplify the techniques discussed. A survey is described that will be distributed to evaluate the final implementation of the renderer.

1. Introduction

Physically Based Rendering (PBR) is the practice of simulating the physics of light and matter interaction to achieve photorealistic shading. When working with empirical lighting models, such as the *Phong reflectance mode*l (Phong, 1975), artists must reconfigure materials to physically plausible results in different lighting environments. PBR techniques are used to create materials which are physically correct in all lighting conditions. (Pranckevičius, 2014). PBR is now extremely prevalent in real-time and offline rendering (Pharr et al., 2017).

The objective of this research is to develop a real-time application that exemplifies the techniques discussed. It will render complex physically-based models and allow the user to configure the lighting conditions. A survey will be taken to determine the effectiveness of the implementation, asking participants to evaluate a number of rendered images for their photorealism. One of these renders will be from the new renderer, one from a basic empirical renderer produced earlier, and two from commercial game engines. The participant will not know what was used to render each image.

1.2. Ethical Considerations

This research will include a survey in order to record how effective the renderer is at producing physically plausible images. The data collected will then be analysed to evaluate the effectiveness of the solution. The feedback will be recorded anonymously. Subjects must agree to have data collected on their responses. Fabricating data in research is an ethical violation. The intellectual property of all third-party resources used must be honoured by giving appropriate credits.

1.3. Justification

Consumers are constantly demanding higher graphical fidelity. The quality of a game's graphics is the most important factor influencing a consumer's decision to buy a computer game (Entertainment Software Association, 2017). Modern game engines, including Frostbite (Lagarde and Rousiers, 2014), Unity (Pranckevičius, 2014), and Unreal (Karis, 2013), employ physically-based rendering.

2. Theoretical Background

Empirical lighting models are computationally cheap and produce visually appealing results, but achieving greater realism requires creating a physically 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 interactions of light and matter requires a basic understanding of the nature of light.

2.1. Mathematical Notation

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

2.2. Radiometry

Radiometry is the core field concerned with measuring electromagnetic radiation. Light is modelled as an electromagnetic transverse wave - a wave that oscillates the magnetic and electric fields perpendicularly to the direction of its propagation (Hoffman, 2013). The oscillations of both fields are coupled.

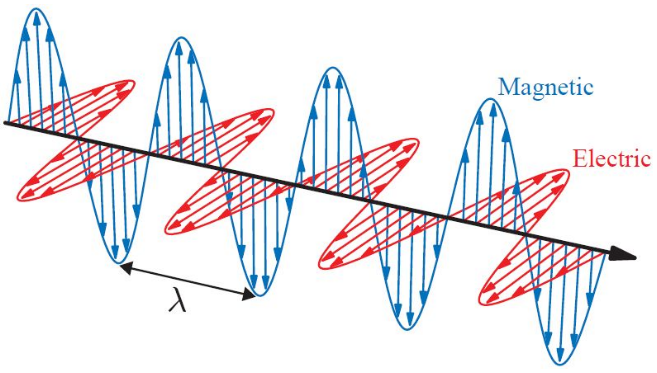


Figure 2.1. Light, an electromagnetic transverse wave (Akenine-Möller et al., 2018).

Figure 2.1 depicts an electromagnetic wave. This is the simplest wave possible; a sine function. It has one wavelength, denoted with λ. Waves with different wavelengths tend to have different properties. The perceived colour of light is strongly related to its wavelength. Light with a single wavelength is *monochromatic*. Most light waves are *polychromatic*, containing many different wavelengths. Human vision can only comprehend a subset of this range, from 400 nanometers for violet light, to 700 nanometers for red light. This tiny part of the range is of interest for shading (Hoffman, 2013). This range is visible in Figure 2.2.

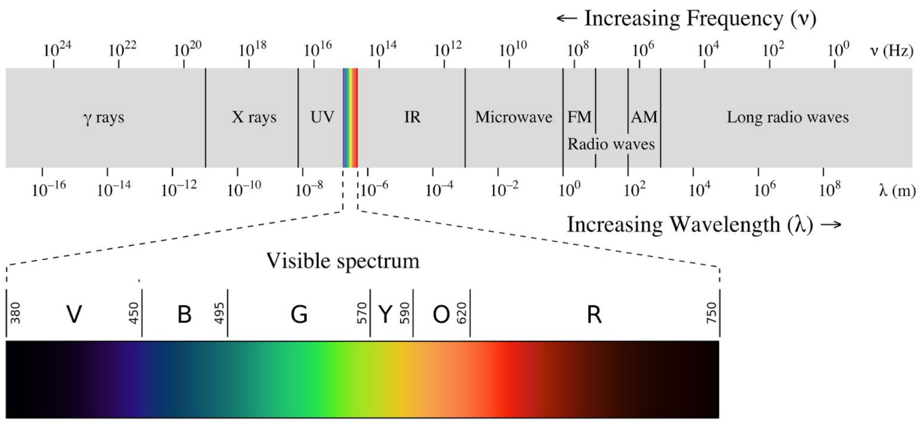

← Increasing Frequency (ν)
10^24 10^22 10^20 10^18 10^16 10^14 10^12 10^10 10^8 10^6 10^4 10^2 10^0 ν (Hz)
γ rays
X rays
UV
IR
Microwave
FM
AM
Radio waves
Long radio waves
10^−16 10^−14 10^−12 10^−10 10^−8 10^−6 10^−4 10^−2 10^0 10^2 10^4 10^6 10^8 λ (m)
Increasing Wavelength (λ) →
Visible spectrum
380 V 450 B 495 G 570 Y 590 O 620 R 750

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

2.2.1. Basic Quantities

There are four radiometric quantities of interest in shading: *flux*, *irradiance / radiant exitance*, *intensity*, and *radiance* (Pharr et al., 2017).

2.2.1.1. Energy

Taking successive limits over time and direction allows radiometric quantities to be derived from energy. Sources of illumination emit photons. Each photon carries a specific amount of energy and exists at a specific wavelength. Energy is measured in *joules, J*. The following quantities can be thought of as different ways of measuring photons. The amount of energy carried by a photon can be calculated:

$$Q = \frac{hc}{\lambda}. \quad (2.1)$$

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

2.2.1.2. Flux

Energy units typically express energy over time. *Radiant flux*, *Φ,* is the total energy passing through a surface per unit time. Radiant flux can be calculated by taking the limit of differential energy per differential time (Pharr et al., 2017):

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

Flux is measured in *watts, w*. Flux is also known as *power*. Given flux as a function of time, it is possible to integrate over a range of times to compute the total energy:

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

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.

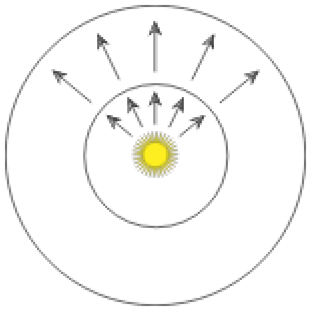


Figure 2.3. Radiant flux, being measured at spheres surrounding a point light (Pharr et al., 2017)

2.2.1.3. Irradiance

To measure flux, an area is required over which photons per time can be observed. The average density of radiant flux over the finite area *A* can be calculated by

$$E = \Phi / A. \quad (2.4)$$

When measuring the area density of flux arriving at a surface, this quantity is *irradiance, E*. When measuring the area density of flux leaving a surface, this quantity is *radiant exitance,* *M*. The term irradiance is commonly used to refer to both cases. Pharr (2017) prefers to use different terms for each case for clarity. 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 uniformly in all directions, then for a sphere with radius *r*,

$$E = \frac{\Phi}{4\pi r^2}. \quad (2.5)$$

This is why the energy received from a point light source becomes less intense with the squared distance from the light. Irradiance can be defined by taking the limit of differential power per differential area at **p**:

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

Irradiance can be integrated over to find radiant flux:

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

2.2.1.4. 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 (Akenine-Möller et al., 2018). 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).

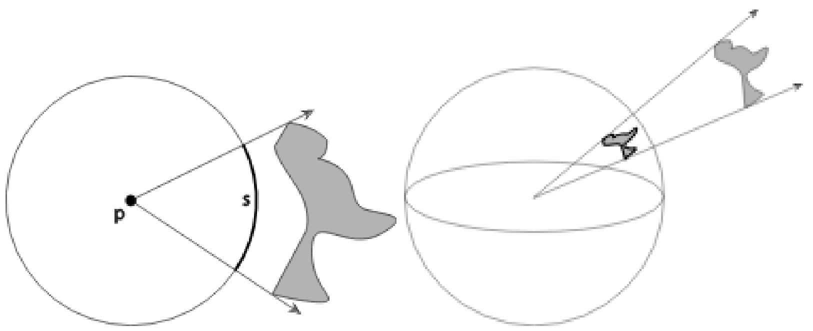


Figure 2.4. Planar angle and solid angle (Pharr et al., 2017)

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

$$I = \frac{\Phi}{4\pi r^2}. \quad (2.8)$$

To take the limit of a differential cone of directions:

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

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

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

2.2.1.5. Radiance

Radiance measures irradiance with respect to solid angles (Lengyel, 2012). Radiance is defined by:

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

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 (Pharr et al., 2017). To define radiance in terms of flux:

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

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, respectfully (Akenine-Möller et al., 2018). The direction vector, denoted by **d**, indicates the direction of the ray, which faces away from **x** by convention.

2.3 Surface Reflection

2.3.1. The BRDF

The traditional Phong reflectance model consists of three components: ambient, diffuse and specular light (Phong, 1975). Ambient light provides a background level of illumination and is reflected equally in all directions by everything in the scene. The diffuse and specular components are directional in nature, reflecting light relative to the position and illumination of a particular light source. Diffuse light is modelled as perfectly Lambertian reflection from a surface that is scattered in all directions. Specular light is modelled as mirror-like highlights that are 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 the way in which the radiant energy contained 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. The reflected light is usually scattered in every direction in a non-uniform distribution. PBR renderers use BRDFs that produce a physically correct distribution of reflected light.

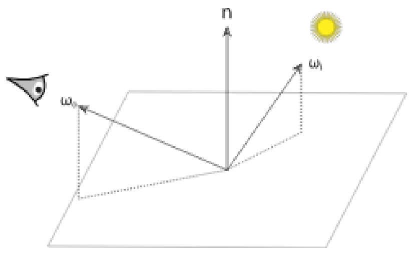


Figure 2.5. A BRDF where ω_i is **l** and ω_o is **v** (Pharr et al., 2017)

Consider Figure 2.5: here, the BRDF is used to describe how much light along **l** is scattered from the surface in the direction **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 a requirement of offline rendering algorithms such as bidirectional path tracing. In practice, real-time BRDFs often violate reciprocity without noticeable artefacts (Akenine-Möller et al., 2018). Additionally, in real-time rendering, exact energy conservation is not strictly necessary, but an approximation of energy conservation is important to achieving realistic lighting (Akenine-Möller et al., 2018).

Originally, BRDFs were 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 (due to blemishes or surface details that change the reflective properties) is called a *spatially varying BRDF* (SVBRDF) or *spatial BRDF* (SBRDF). This commonly means a combination of BRDF and texture mapping (McAllister, 2004). This case is exceedingly common, however, and most publications use the term BRDF to implicitly 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:*

$$L_o(\mathbf{p}, \mathbf{v}) = \int_{\mathbf{l} \in \Omega} f(\mathbf{l}, \mathbf{v}) L_i(\mathbf{p}, \mathbf{l})(\mathbf{n} \cdot \mathbf{l}) d\mathbf{l}. \quad (2.14)$$

The $\mathbf{l} \in \Omega$ subscript on the integral means that the function integrates over **l** vectors that lie within the unit hemisphere above the surface, which is centred on the surface normal **n**. The light direction **l** is continuously swept over the hemisphere of incoming directions. Any incoming light direction can and likely 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 the reflectance equation (Equation 2.14) 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 locally 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 (Akenine-Möller et al., 2018).

2.3.1. 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**. For this reason, the notation $F(\mathbf{n}, \mathbf{l})$ is also used. This function has the following characteristics:

1. *Normal incidence*: when $\theta_i = 0^\circ$, 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 $\theta_i = 90^\circ$. (See Figure 2.6)

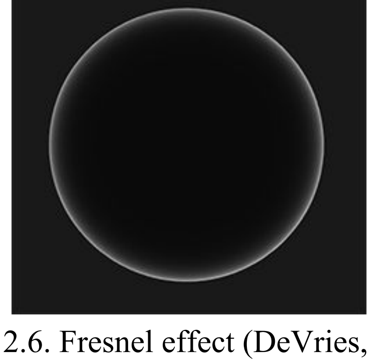


Figure 2.6. Fresnel effect (DeVries, 2016c)

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

2.3.2. Microgeometry

A physically-based BRDF must compensate for microscopic imperfections. These irregularities are so small that a renderer cannot feasibly render them. It is the aggregate behaviour 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 a *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 accurately in the same direction, resulting in sharper reflections. This is the primary effect of microgeometry.

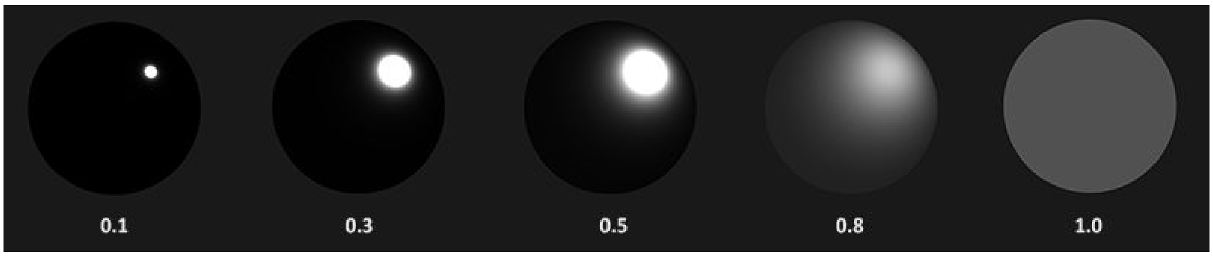


Figure 2.7. The visual result of increasing roughness with a normal distribution function (DeVries, 2016c)

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. The geometry function takes the roughness parameter and calculates the amount of shadowing and masking, with rough surfaces having a higher probability of both.

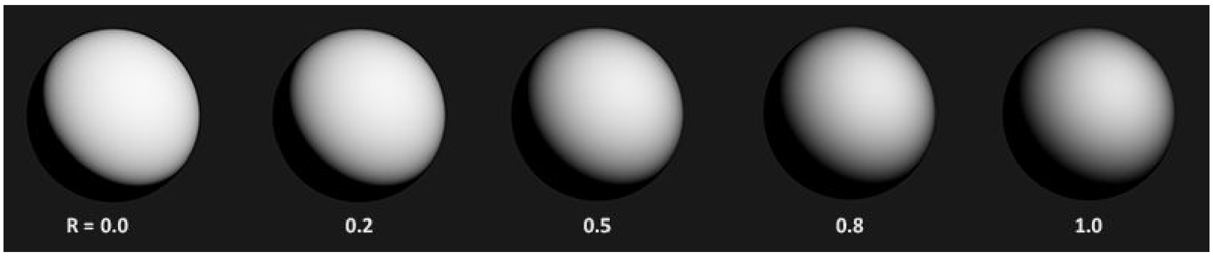


Figure 2.8. The visual result of increasing roughness with a geometry function (DeVries, 2016c)

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

$$f(\mathbf{l},\mathbf{v}) = f_d(\mathbf{l},\mathbf{v}) + \frac{D(\mathbf{h})F(\mathbf{v},\mathbf{h})G(\mathbf{l},\mathbf{v},\mathbf{h})}{4(\mathbf{n}\cdot\mathbf{l})(\mathbf{n}\cdot\mathbf{v})}. \quad (2.15)$$

Where $f_d(\mathbf{l}, \mathbf{v})$ is the diffuse BRDF (Lambertian reflection is the most common). The specular component of this equation is the general *Cook-Torrance* microfacet specular BRDF (Cook and Torrance, 1982). *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 section 3.2. Creating an effective physically-based BRDF requires that these different terms are evaluated.

3. Literature Review

3.1. Diffuse BRDF

The majority of this research is focused on finding a good specular BRDF, however, a number of physically based diffuse BRDFs have been presented to model the diffuse component more accurately than a typical Lambertian approach.

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). A Lambertian model assumes that scattered light loses all directionality, allowing constant diffuse reflectance (Burley, 2012). Where c_{diff} is the diffuse albedo colour of the material, the Lambertian BRDF is defined:

$$f_d(\mathbf{l}, \mathbf{v}) = \frac{c_{\text{diff}}}{\pi}. \quad (3.1)$$

Lambertian diffuse is not always physically accurate. A surface that conforms with Lambert's Law is equally bright from all viewing directions. This is an inadequate approximation for a number of real-world materials with rough surfaces (Oren and Neyar, 1994). However, when used in conjunction with a physically-based specular BRDF, the visual difference is minimal (see Figure 3.1). For this reason, Epic Games chose to continue using Lambertian diffuse BRDF in their renderer (Karis, 2013).

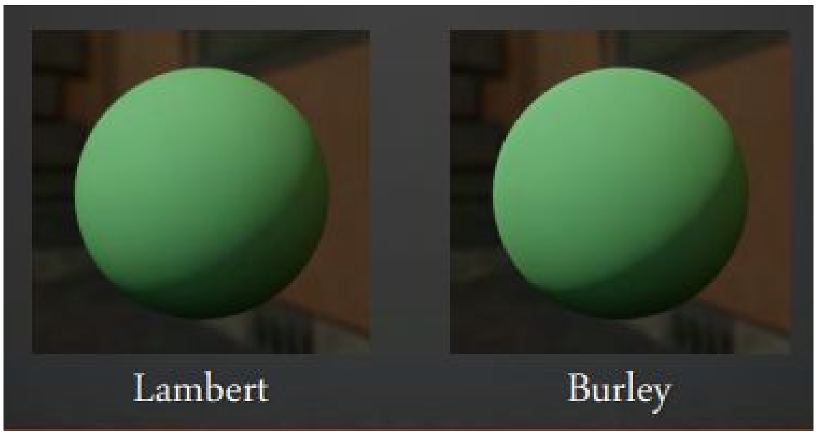


Figure 3.1. Comparison of Lambertian diffuse and Disney diffuse model (Burley, 2012)

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 decided to model surfaces as a collection of perfectly Lambertian microfacets (Oren and Neyar, 1994). It combines Lambertian diffuse with the Torrance-Sparrow model, which assumes the surface is composed of long, symmetric V-cavities, where each V-cavity has two opposing planar facets (Torrance and Sparrow, 1967). The model is able to account for geometric phenomena that Lambert ignores, such as shadowing, masking, and interreflections between facets (Oren and Nayar, 1994).

3.1.3. Disney (Burley)

Disney considered using Lambertian diffuse and Oren-Nayar but found that neither were able to meet their measured data (Burley, 2012). For their principled BRDF, they presented a new diffuse BRDF that transitions between Fresnel shadow for smooth surfaces and ads a highlight for rough surfaces. They found it to be artistically 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 (Lagarde and Rousiers, 2014). When developing UE4, Epic Games considered the Burley diffuse BRDF but considered it to be too expensive to calculate for the small increase in realism (see Figure 3.1). The Burley diffuse model is used in the Frostbite game engine, adjusted with energy renormalization. (Lagarde and Rousiers, 2014).

3.2. Microfacet Specular BRDF

The purpose of this research is to develop a PBR renderer that is comparable in quality to that of a commercial game engine. Both Frostbite and Unreal use a similar microfacet specular BRDF, combining a GGX normal distribution function with a Smith-correlated geometry term and Fresnel-Schlick fresnel approximation (Lagarde and Rousiers, 2014; Karis, 2013). For this reason, these terms were given more attention than alternative solutions.

3.2.1. Specular D

GGX is arguably the most common NDF used in real-time and offline rendering (Akenine-Möller et al., 2018). Originally introduced by Trowbridge and Reitz (1975), this distribution function was independently discovered by Walter et. al (2008) who named it *GGX*. The GGX distribution is:

$$D(\mathbf{m}) = \frac{\chi^{+}(\mathbf{n}\cdot\mathbf{m})\alpha_g^2}{\pi(1+(\mathbf{n}\cdot\mathbf{m})^2(\alpha_g^2-1))^2}. \tag{3.2}$$

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).

In GLSL, a GGX NDF can be defined:

```
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;
}
```

3.2.2. Specular G

Heitz (2014) has demonstrated that the only two geometry functions that are mathematically 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 behaviour of random microsurfaces (Akenine-Möller et al., 2018). The Smith visibility function Smith's method takes into account the view direction for geometry obstruction and the light direction for geometry shadowing:

$$G_2(\mathbf{n},\mathbf{v},\mathbf{l}) = G_1(\mathbf{n},\mathbf{v})G_1(\mathbf{n},\mathbf{l}). \quad (3.3)$$

There are numerous forms of the Smith G_1 function. Disney uses the Smith model derived by Walter et. al (2008) for GGX with minor modifications. This function was also favoured by Lagarde and Rousiers (2014). 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:

$$G_1(\mathbf{s}) \approx \frac{\mathbf{n}\cdot\mathbf{s}}{(\mathbf{n}\cdot\mathbf{s})(1-k)+k}. \quad (3.4)$$

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):

$$k_{direct} = \frac{(\alpha+1)^2}{8}. \quad (3.5)$$

$$k_{IBL} = \frac{\alpha^2}{2}. \quad (3.6)$$

Image-based lighting is explored further in section 3.3. This approximation is commonly referred to as Schlick-GGX. A Schlick-GGX geometry function can be defined in GLSL:

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

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

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

    return ggx1 * ggx2;
}
```

3.2.3. Specular F

The Fresnel-Schlick function is a computationally inexpensive approximation that is reasonably accurate (Schlick, 1994). 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:

$$F(\mathbf{n}, \mathbf{l}) = F_0 + (1 - F_0)(1 - (\mathbf{h} \cdot \mathbf{v}))^5. \quad (3.7)$$

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 physically plausible result without adding complex surface parameters. For Unreal, this value is 0.04 (Karis, 2013). If a surface is dielectric, F_0 is used, otherwise, the surface colour is used. This technique is possible because metallic surfaces have no diffuse reflections (Akenine-Möller et al., 2018). The *metalness* of a material can be defined as a surface parameter, as a simple floating point number between 0 for dielectric materials, and 1 for metals (Burley, 2012).

A Fresnel-Schlick based Fresnel function can be defined in GLSL:

```
vec3 f0 = vec3(0.04f);
f0 = mix(f0, surface.rgb, metalness);

vec3 f_schlick(float cos_theta, vec3 f0)
{
    return f0 + (1.0f - f0) * pow(1.0f - cos_theta, 5.0f);
}
```

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; Pranckevicius, 2015). Approximating the integral of incoming light in all directions cannot be easily achieved in real time. It is possible to pre-calculate the integral of incoming light and store it for later use at runtime. IBL techniques capture the lighting environment from all directions around a given point and use it to evaluate general BRDFs (Akenine-Möller et al., 2018). When using IBL techniques to solve simulate diffuse and specular effects, environment maps are known as *diffuse light probes* and *specular light probes*, respectfully (Spogreev, 2016; Akenine-Möller et al., 2018).

Bjorke (2004) demonstrates how to achieve localised reflections when using cube maps for image-based lighting, avoiding the typical limitation of cube map reflections where they appear infinitely far away. Spogreev (2016) demonstrates how light probes can be generated when loading a scene in order to achieve dynamic, physically-based environment lighting. The scene features a number of light probes, a combination of diffuse and specular light probes. Objects in the scene receive lighting based on proximity to nearby probes.

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 very 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 produced from a sample of the surrounding lighting environment and are blurred heavily, which means they can be stored at low resolution. The convoluted result is stored in each texel of a cube map. Sampling this cube map in any direction will yield the scene's irradiance in that direction. Instead of the reflected view direction being used to sample the cube map, the surface normal is used. It is also possible to model the fresnel effect, increasing specular reflectance at glancing angles (Lagarde and Rousiers, 2014).

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, taking a significant amount of time to process high-resolution environment maps. Offloading this work to the GPU is a more favourable decision (Spogreev, 2016). Irradiance can be calculated by integrating the incoming light and summing the cosine-weighted contribution of all light sources affecting the surface in a given normal direction (Akenine-Möller et al., 2018). It is also possible to integrate and project the integral onto 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, because 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).

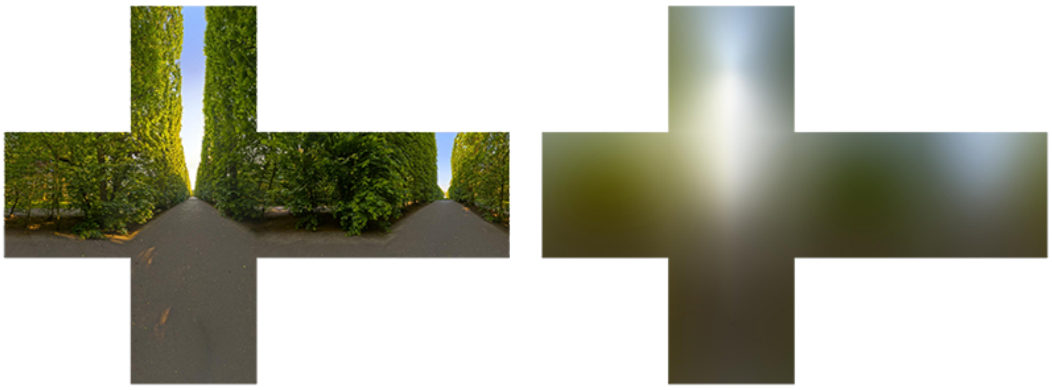


Figure 3.2. Environment map and its resulting irradiance environment map (DeVries, 2016a)

Once convoluted, the irradiance map can be incorporated in a GLSL shader to calculate the ambient colour:

```
vec3 ambient = texture(irradiance_map, n).rgb;
vec3 ks = f_schlick(max(dot(n, v), 0.0), f0);
vec3 kd = 1.0 - ks;
vec3 irradiance = texture(irradiance_map, n).rgb;
vec3 diffuse    = irradiance * albedo;
vec3 ambient    = (kd * diffuse) * ao;
```

Roughness is not taken into account in this shader, which means the reflectivity of the surface will always be high. Lagarde and Rousiers (2014) demonstrate a generalised form of Fresnel-Schlick that provides more artistic control over the reflection:

```
vec3 f_schlick(float cos_theta, vec3 f0, float f90)
{
    return f0 + (f90 - f0) * pow(1.0 - cos_theta, 5.0f);
}
```

It is now possible to alleviate the reflectivity issue by incorporating surface roughness when computing the fresnel response.

```
vec3 ambient = texture(irradiance_map, n).rgb;
vec3 ks = f_schlick(max(dot(n, v), 0.0), f0, roughness);
vec3 kd = 1.0 - ks;
vec3 irradiance = texture(irradiance_map, n).rgb;
vec3 diffuse    = irradiance * albedo;
vec3 ambient    = (kd * diffuse) * ao;
```

3.3.2. Specular Light Probes

Specular light probes capture the radiance at a given point in the scene from all directions in a cube map, also known as a *specular cube map* (Akenine-Möller et al., 2018). The information stored in the light probe is then used to evaluate BRDFs. Numerous 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:

$$\int_{\mathbf{l}\in\Omega} f_r(\mathbf{l},\mathbf{v})L_i(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l} \approx \int_{\mathbf{l}\in\Omega} D(\mathbf{r})L_i(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l} \int_{\mathbf{l}\in\Omega} f_r(\mathbf{l},\mathbf{v})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l}. \tag{3.8}$$

This technique separates the specular work into two parts that can be convoluted separately and then combined to achieve indirect specular image-based lighting. Lagarde and Rousiers (2014) refer to the first term as the *LD* term, and the second term as the *DFG* term. LD must be computed for each light probe separately, whereas DFG can be computed once and reused for all light probes.

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, the convolution input of specular image-based lighting is an HDR environment map. The environment map is convoluted with more scattered samples for increasing roughness levels, creating blurred reflections. Each level convoluted is stored in the mipmap levels of a cube map, decreasing the resolution needed to store each subsequent roughness level (see Figure 3.3).

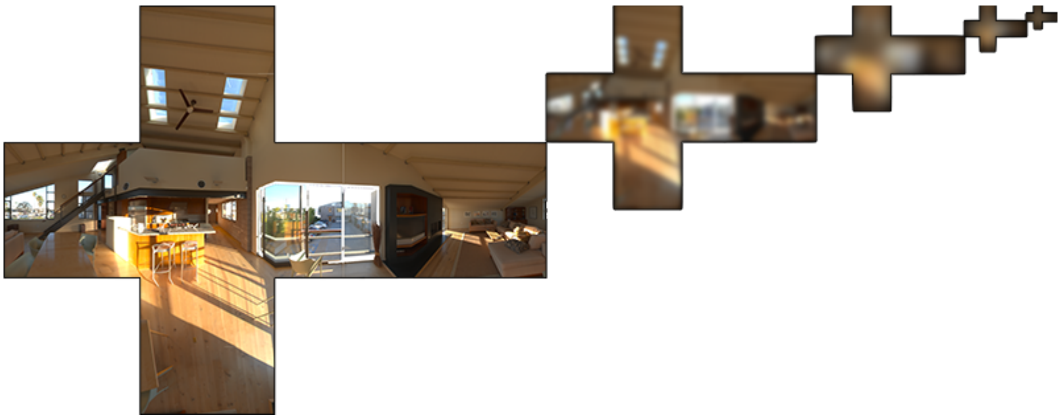


Figure 3.3. Pre-filtered environment map with filtered roughness levels stored in each mipmap level (DeVries, 2016b)

The diffuse BRDF can be easily integrated because it relies only on the normal and light directions. Specular BRDFs rely on many variables. Pre-integrating the integral 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 efficiently pre-integrated, with the assumption that the BRDF shape is isotropic at all view angles. This compromise prevents introduces error. Figure 3.4 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* is used to capture 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.

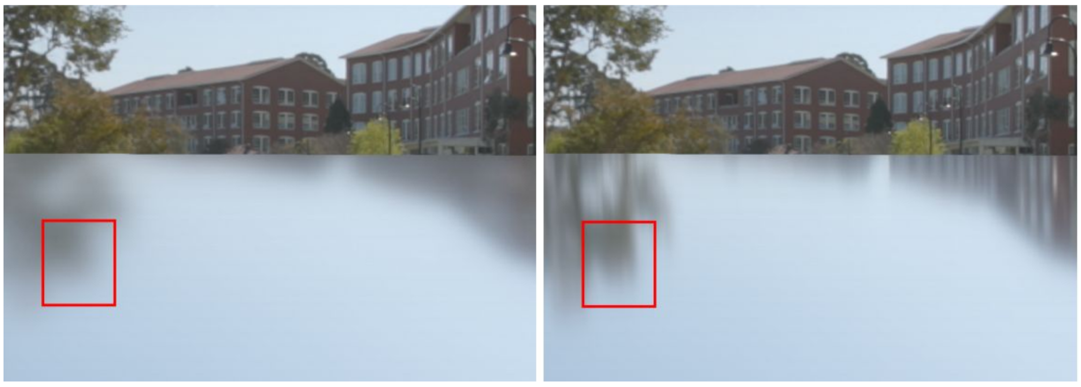


Figure 3.4. Loss of stretched specular reflections at grazing angles (Lagarde and Rousiers, 2014)

3.3.2.2. Environment BRDF

The second integral includes everything else. Karis (2013) describes it as integrating the specular BRDF with a solid white environment. Given a roughness value and an angle between the normal and the light direction, it is possible to pre-calculate the result of the BRDF. Karis stores the roughness parameter and the angle in a *lookup texture* (LUT). This LUT is known as a *BRDF integration map,* which can be seen in Figure 3.5. The LUT outputs a scale and a bias for use in the surface's Fresnel response. The Y coordinates map to the roughness, while the X coordinates map to $n \cdot v$. The red component is the scale and the green component is the bias.

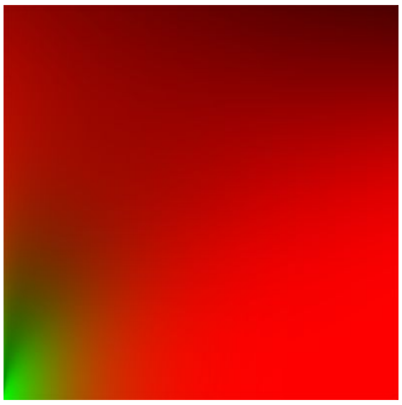


Figure 3.5. BRDF integration map (Karis, 2013)

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

New material properties are required to adapt to a PBR workflow. PBR assets require albedo, metallic, and roughness maps to define the material properties on a per-texel basis (Karis, 2013). Normal maps and ambient occlusion maps are also typical for PBR materials.

Albedo maps can define the base colour of a dielectric surface or the base reflectivity of a metallic surface. The role of an albedo texture is similar to that of a diffuse texture, however, albedo maps deliberately avoid containing any lighting information. Ambient occlusion maps can be used to introduce small-scale shadow detail.

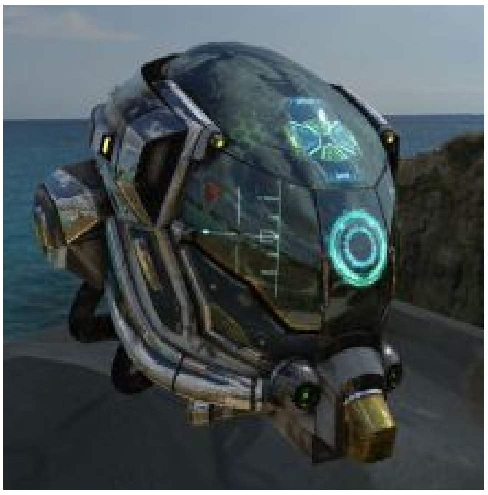
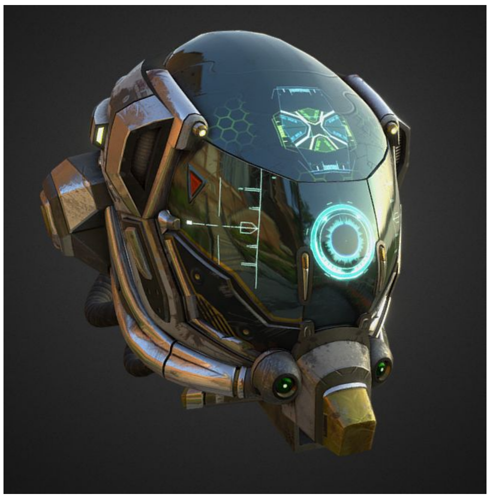
Metallic maps are grayscale images that define if the surface is dielectric or metal using values between 0.0f and 1.0f, respectively. Roughness maps are grayscale images allowing the roughness to be defined between 0.0f and 1.0f. Normal maps define tangent space surface normals per-texel, allowing creating the illusion of surface details where there are none. Ambient Occlusion maps add extra shadowing detail to the surface and surrounding geometry.

4. Proposed Research Methodology

This research will lead on to the implementation and evaluation of a renderer. In order to determine if the produced renderer is effective, surveys will be distributed in which subjects will be asked to rate a series of rendered images for realism between 0 and 10. The results will then be analysed. If the new renderer is ranked similarly to commercial game engines, the solution will be considered a success.

The following is an example survey for a pair of rendered images of the same glTF asset. The final survey will be composed of many of these:

Please rate the following images on a scale of 1 - 10 for their photorealism, where 1 is highly unrealistic and 10 is highly realistic.

5. Technical Overview

This section will detail the requirements, development plan and design of a new PBR renderer, *Moka*.

5.1. Development Tools

5.1.1. Programming Languages

Moka will be developed using C++17, with MSVC as the main supported compiler. The codebase will be designed with extensibility in mind, but to keep the project on schedule only a Windows / OpenGL version of the Moka project will be produced.

Shaders will be 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 will only support a GLSL-based implementation to ensure the project can run on university hardware.

5.1.2. Build System

The project will use CMake so that it will be decoupled from any native build system. CMake will also significantly simplify the process of integrating with third-party libraries due to its ubiquity.

5.1.3. Version Control

The project will make extensive use of Git for version control, with the repository hosted on GitHub. The GitFlow workflow will be used to organise branching. *Posh-Git* is used in a Windows development environment to access Git through a Powershell terminal. When large-scale merges become necessary, GitKraken will be used for its clear graphical interface.

5.1.4. Package Management

Downloading, building and linking to third-party libraries is normally a tedious process in C++, but a number of package managers have emerged that simplify the process. There are now a number of contending package managers to consider, including *Hunter*, *Conan*, and *vcPkg*. The project will make use of a package manager to reduce the work needed to make use of the numerous third-party libraries necessary. These tools will be tested during development.

5.1.5. Dependencies

GLM will be used as the project's maths library (Riccio, 2018). Unfortunately, math types become extremely pervasive through graphics codebases so it will be impossible to abstract this dependency. Though it means committing to a third-party dependency across the codebase, there would be little benefit from spending the time to make a maths library of equivalent depth and functionality. Moka will commit to GLM as the primary maths library for this reason. GLM can be replaced with a bespoke maths library in the future.

For windowing and input handling, *SDL2* will be used (Lantinga, 2018). SDL2 prevents 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 will significantly speed up the development of windowing and input handling.

Spdlog will be 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 the glTF model format (Fujita, 2018). Moka will make use of it to import the many test PBR models made available by Khronos Group.

Dear ImGui will be used for developing simple user interfaces to allow configuration of the rendering environment (Cornut, 2018).

5.2. General Constraints

5.2.1. Availability of Hardware / Software

There are a number of limitations or constraints to consider that will have an effect of the design on the system. Most of these constraints concern hardware and software availability. Though the software will be designed to be portable, only a Windows / OpenGL build will be produced. This means that the software will only run on a Windows machine with an OpenGL version 3.3+ enabled graphics device. There are also restrictions on the development environment that can be used to compile the software. The machine must have CMake 3.2+ installed to generate solution files. They must also have a C++17-compliant compiler. Moka will make use of new language features and libraries that are not yet available in Visual Studio by default, such as *std::filesystem*, *structured bindings* and *if constexpr*. The CMakeLists will specify the "/std:c++latest" compiler argument necessary to enable these features.

5.2.2. Availability of PBR Assets

A significant obstacle is the availability of PBR 3D models. Moka must be able to import and render 3D assets to prove the effectiveness of the PBR implementation, and these assets must be designed for use in PBR rendering. PBR requires multiple specialised texture maps to define the material properties. Model formats like OBJ and FBX do not define these properties, only having standardised properties for more common texture map uses. For this reason, model importers must prepare PBR models using these formats using the available material properties and map them to the desired PBR texture maps when importing them. This is not ideal as it requires additional work to prepare assets in a non-standard way to get them to import properly. However, 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 also 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 also available in most 3D modelling 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 will use these assets to test Moka's implementation of PBR.

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 will only be importing PBR-ready glTF models, there is no reason to commit to a large asset importing library such as Assimp. Moka will make use of tinyglTF for this reason. It is a simple header-only library for reading glTF 2.0 assets.

5.2.3. Shader Permutations

Importing assets authored by third parties bring additional complexity - each asset may define a number of materials and each may be vastly different from the last. For Moka to be able to import and render assets in a uniform, generic way without requiring modifications to the asset, it must have a way to deal with shader permutations. When loading an asset, materials must be attached to a shader that is written to expect those exact inputs.

Moka will feature a simple, automatic system for dealing with shader permutations. Instead of evaluating material inputs at runtime, Moka will evaluate the inputs at compile time. As materials are processed, a description of the material will be built, detailing all the material inputs and their uses. Moka will hash the material description and maintain a lookup table of currently loaded shaders, using the hashed value as a key. At the end of the primitive importing process, Moka will perform 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 will compile all the uniforms and their uses in computing the final colour result.

5.3. Implementation Goals

The style used by the Moka project will be derived from Boost. All C++ code produced will fight for clarity and correctness. Optimisation of the code is a secondary concern. Moka will make 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 will be followed. The naming convention of the C++ standard library will be followed:

- All names will 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 will be valued above brevity. Exceptions will be used for error reporting where appropriate. Sample programs will be provided to demonstrate the use of the library. Four spaces will be used instead of tabs. Line lengths will be limited to 80 characters. All source files will begin with:

- A comment line describing the contents of the file.
- A comment line referencing the Moka GitHub repository.
- A comment line directing users to a copy of this research paper.

Moka will be designed with simplicity in mind. Any features that will not actively contribute towards creating a PBR renderer will be considered a 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 will be focused on creating a simple, working rendering abstraction over a complete game engine. This ideal will also apply to the design of the user-facing interfaces. Simple and intuitive abstractions should be offered to the programmer, allowing them to use PBR with no need to deal with low-level native APIs.

The code should have a well-documented user-facing API, featuring Doxygen-style code comments detailing the responsibilities of each publicly-facing facet of the framework.

5.4. System Overview

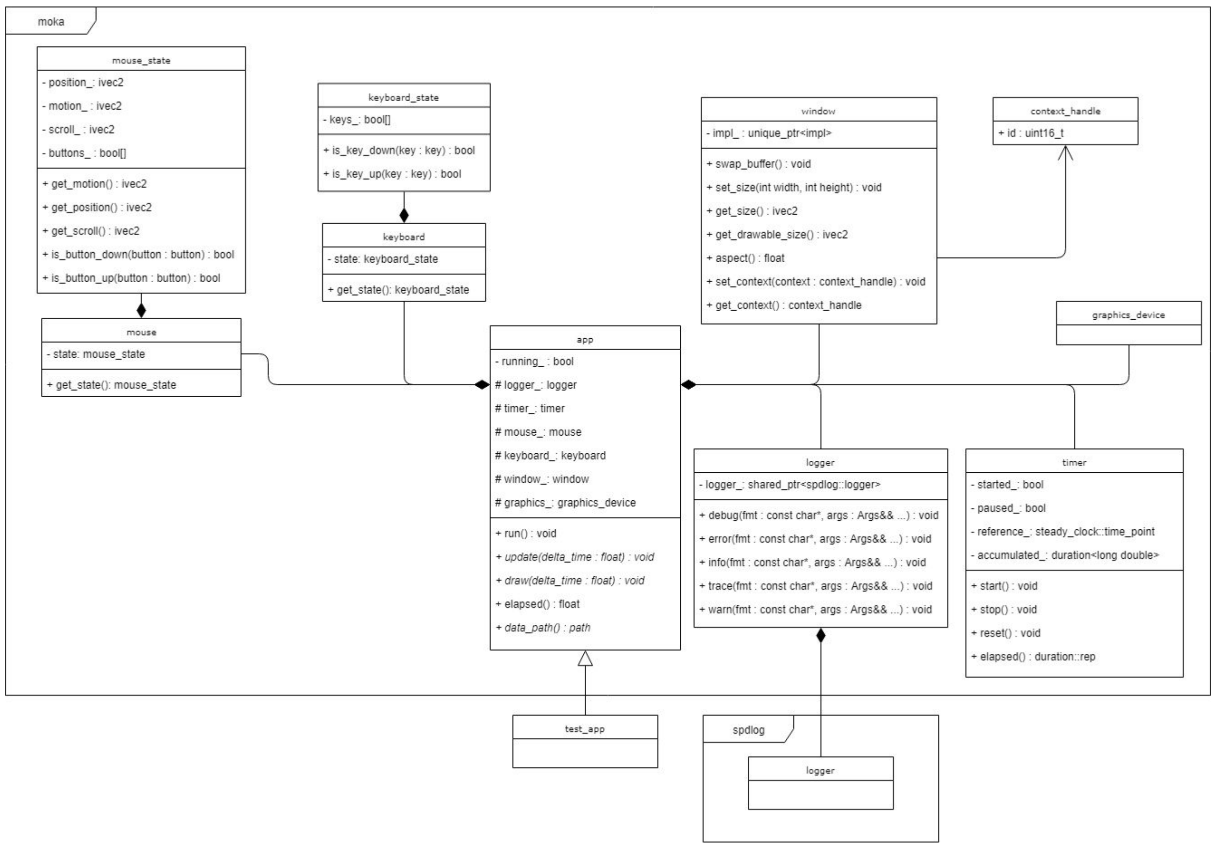
Moka will be a framework for creating Physically Based Rendering applications. It will define a library offering a high level of abstraction that allows complex PBR rendering to be performed with relative ease. The application-facing interface will be agnostic to any native rendering API / window system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

Moka will present an interface for stateless, multithreaded rendering. Unlike OpenGL, any state set for rendering one primitive will not affect subsequent operations. Additionally, draw calls made from the application will not be rendered immediately - they will be added to a buffer that is then processed and rendered later by a dedicated rendering thread.

Before rendering can happen, Moka must initialize the native window system, create a rendering context, initialize the native rendering API, import PBR-ready 3D assets and set up a basic scene. The specific implementation of these features is not relevant to a PBR, so all of this functionality will be abstracted and organised into a supporting library of classes. This library will be similar in design to MonoGame or XNA, presenting useful utilities for quickly creating test applications. The entire application-facing interface will be designed in this fashion, providing a high level of abstraction for creating PBR applications. Wherever access to low-level graphics functionality is exposed, Moka will ensure that the platform-specific implementation of these features is well hidden.

5.4.1. 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.

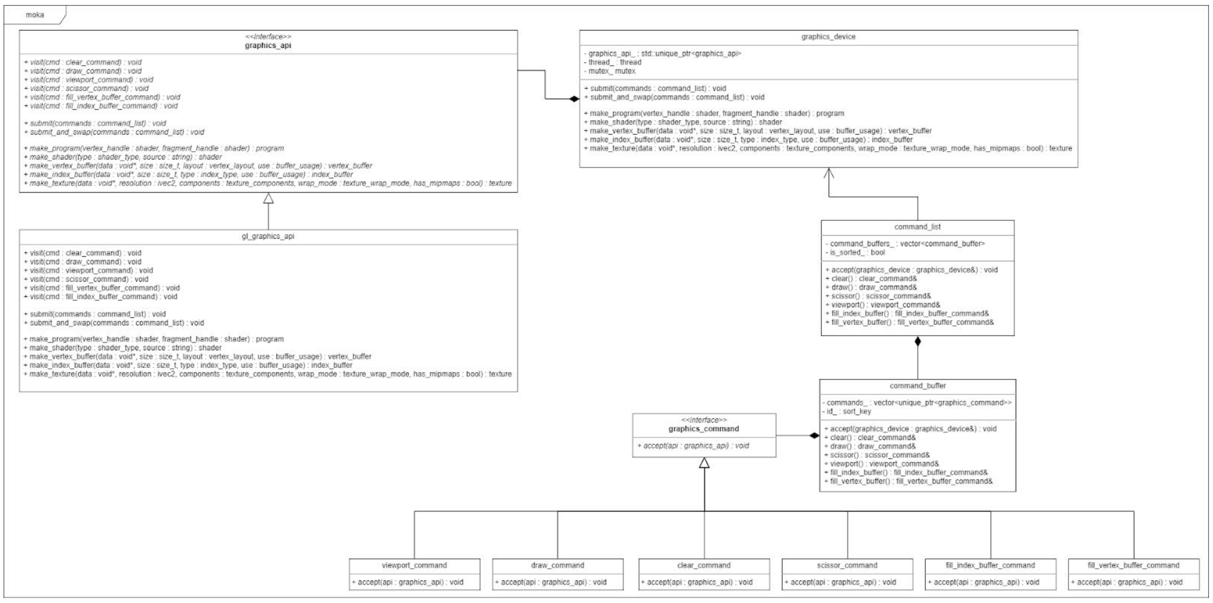


moka
mouse_state
- position_: ivec2
- motion_ : ivec2
- scroll_ : ivec2
- buttons_ : bool[]
+ get_motion() : ivec2
+ get_position() : ivec2
+ get_scroll() : ivec2
+ is_button_down(button : button) : bool
+ is_button_up(button : button) : bool
mouse
- state: mouse_state
+ get_state(): mouse_state
keyboard_state
- keys_: bool[]
+ is_key_down(key : key) : bool
+ is_key_up(key : key) : bool
keyboard
- state: keyboard_state
+ get_state(): keyboard_state
window
- impl_ : unique_ptr<impl>
+ swap_buffer() : void
+ set_size(int width, int height) : void
+ get_size() : ivec2
+ get_drawable_size() : ivec2
+ aspect() : float
+ set_context(context : context_handle) : void
+ get_context() : context_handle
context_handle
+ id : uint16_t
graphics_device
app
- running_ : bool
# logger_: logger
# timer_: timer
# mouse_: mouse
# keyboard_: keyboard
# window_: window
# graphics_: graphics_device
+ run() : void
+ update(delta_time : float) : void
+ draw(delta_time : float) : void
+ elapsed() : float
+ data_path() : path
logger
- logger_: shared_ptr<spdlog::logger>
+ debug(fmt : const char*, args : Args&& ...) : void
+ error(fmt : const char*, args : Args&& ...) : void
+ info(fmt : const char*, args : Args&& ...) : void
+ trace(fmt : const char*, args : Args&& ...) : void
+ warn(fmt : const char*, args : Args&& ...) : void
timer
- started_: bool
- paused_: bool
- reference_: steady_clock::time_point
- accumulated_: duration<long double>
+ start() : void
+ stop() : void
+ reset() : void
+ elapsed() : duration::rep
test_app
spdlog
logger

Starting a Moka application requires programmers to subclass the *app* class. Through the app class, their application will have access to the input handling and graphics utilities. Upon calling run(), the game loop will start. The loop will call the virtual update() and draw() methods, which must be overloaded in their concrete implementation. This is a very similar design to XNA / MonoGame's Game class.

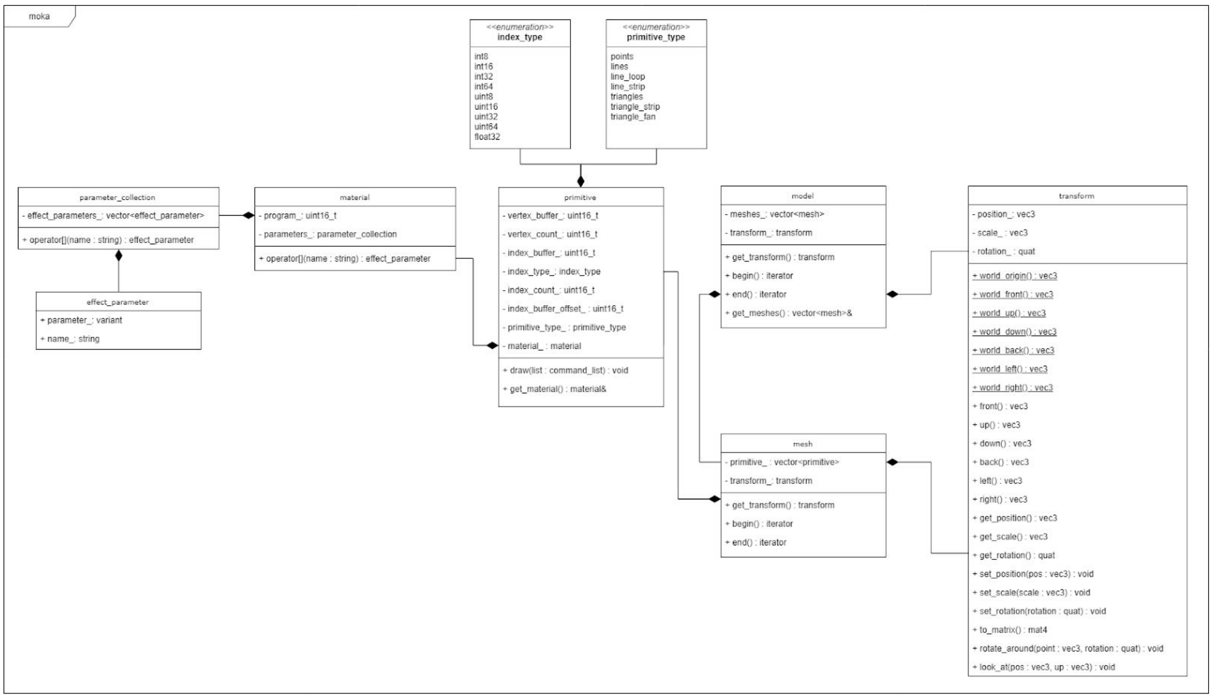
The app class abstracts the native event loop from the programmer and updates the input/rendering utilities appropriately.

5.4.2. Graphics Device


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
+ submit(commands : command_list) : void
+ submit_and_swap(commands : command_list) : void
+ make_program(vertex_handle : shader, fragment_handle : shader) : program
+ make_shader(type : shader_type, source : string) : shader
+ make_vertex_buffer(data : void*, size : size_t, layout : vertex_layout, use : buffer_usage) : vertex_buffer
+ make_index_buffer(data : void*, size : size_t, type : index_type, use : buffer_usage) : index_buffer
+ make_texture(data : void*, resolution : ivec2, components : texture_components, wrap_mode : texture_wrap_mode, has_mipmaps : bool) : texture
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
+ submit(commands : command_list) : void
+ submit_and_swap(commands : command_list) : void
+ make_program(vertex_handle : shader, fragment_handle : shader) : program
+ make_shader(type : shader_type, source : string) : shader
+ make_vertex_buffer(data : void*, size : size_t, layout : vertex_layout, use : buffer_usage) : vertex_buffer
+ make_index_buffer(data : void*, size : size_t, type : index_type, use : buffer_usage) : index_buffer
+ make_texture(data : void*, resolution : ivec2, components : texture_components, wrap_mode : texture_wrap_mode, has_mipmaps : bool) : texture
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 : shader, fragment_handle : shader) : program
+ make_shader(type : shader_type, source : string) : shader
+ make_vertex_buffer(data : void*, size : size_t, layout : vertex_layout, use : buffer_usage) : vertex_buffer
+ make_index_buffer(data : void*, size : size_t, type : index_type, use : buffer_usage) : index_buffer
+ make_texture(data : void*, resolution : ivec2, components : texture_components, wrap_mode : texture_wrap_mode, has_mipmaps : bool) : texture
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&
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&
<<Interface>>
graphics_command
+ accept(api : graphics_api) : void
viewport_command
+ accept(api : graphics_api) : void
draw_command
+ accept(api : graphics_api) : void
clear_command
+ accept(api : graphics_api) : void
scissor_command
+ accept(api : graphics_api) : void
fill_index_buffer_command
+ accept(api : graphics_api) : void
fill_vertex_buffer_command
+ accept(api : graphics_api) : void

The Moka *graphics_device* class is an example of a simple bridge pattern - the renderer backend is a concrete implementation of an abstract class. The Moka application will not interact with OpenGL directly and therefore will be loosely coupled to the native rendering API. The graphics_device class will execute commands on a separate thread from the main application. The main application thread will enqueue 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. 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. A more efficient model for a command buffer would abandon runtime polymorphism and define each command as a POD struct. Then, when adding the commands to the command buffer, copy the memory into a contiguous buffer. However, this will take longer to develop due to the increased boilerplate memory management and will produce code that is potentially harder to reason about.

5.4.3. Model System



The model system is loosely based upon the glTF 2.0 specification. It will likely evolve over development. A model is composed of one or more meshes, which are composed of one or more primitives. Primitives define the vertex buffer, index buffer, and material that is being used to render. The material contains the shader handle and all effect parameters necessary to render. The effect parameter class encapsulates a discriminated union that will be able to hold all shader uniforms necessary to render a material. The transform class is based on that of Unity and is general enough to be used for camera transformation. It features helper functions look_at(), rotate_around() and to_matrix(), which will cut down the amount of boilerplate maths code across test applications.

6. Project Management

6.1. Development Methodology

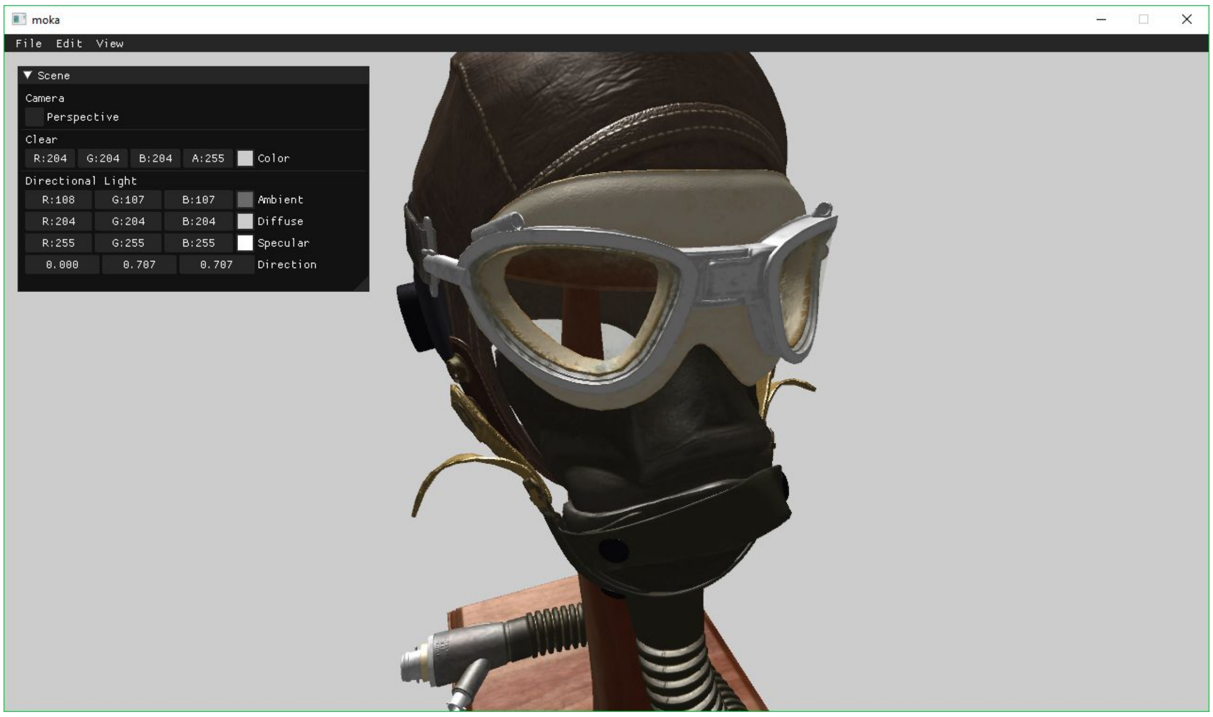
Software development methodologies encourage a systemic approach towards development to improve the quality of the final product. A project of this size and complexity requires forward planning to ensure its success. This project will be implemented alongside performing additional research and experimentation, so it is entirely possible that the project requirements or implementation plans will change due to information discovered during development. This means that a traditional linear project management methodology, such as the Waterfall model, is unsuitable. Constantly rewriting the project schedule to compensate for new obstacles would be very time consuming, which is unacceptable given the time constraints the project already faces.

While the project will not benefit from a rigid linear project plan, deadlines must still be met. For this reason, Moka will be developed using a Scrum-based agile 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. Development will be broken into discrete tasks that can be completed over a period of 2 week 'sprints'. To ensure that the project is progressing as planned, the project will be reassessed at the end of each sprint. It is at this time that the project supervisor will be consulted and the progress made and work remaining will be considered. Any changes to the project will be made as a result of this reassessment.

6.2. Schedule

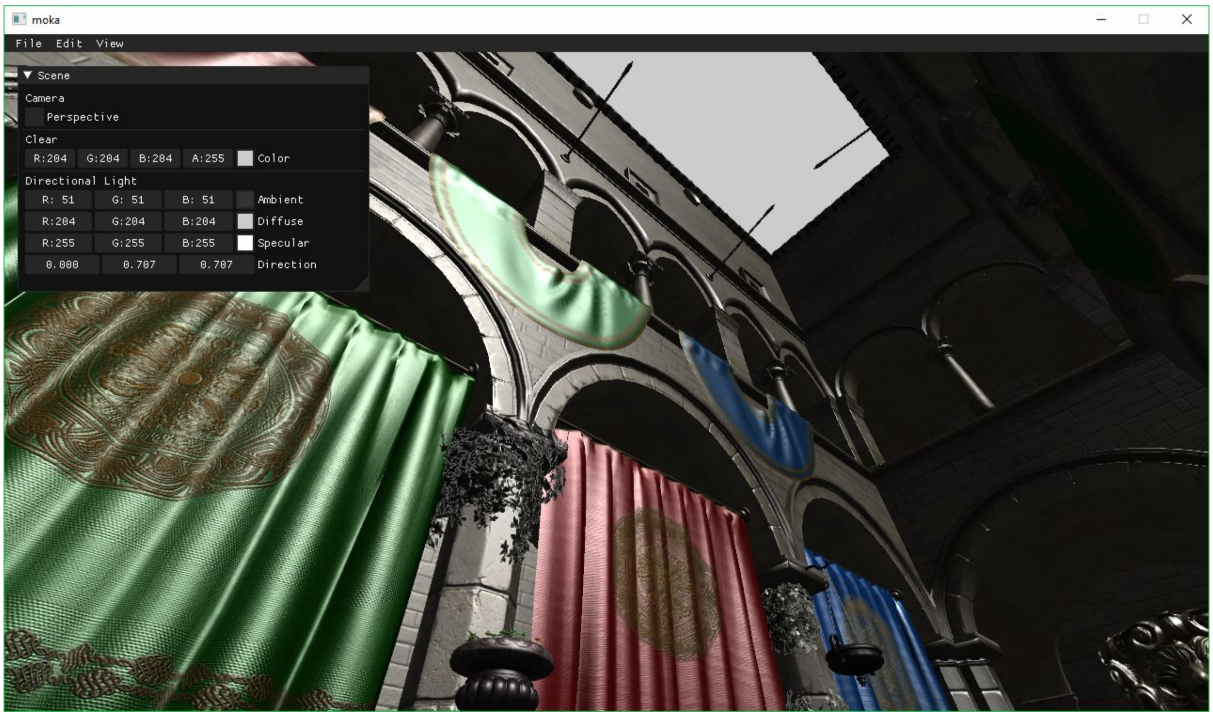
A Gantt chart breaking down the work completed and the current schedule is included in section 11.1. It details the other modules deliverables and takes them into consideration. This chart is incomplete and does not try to set a fixed path for the project's development. This would be unwise as the schedule will almost certainly change due to the demands of other modules or other unforeseen circumstances. The final report will include a chart detailing the real breakdown of work over the year.

7. Current Progress



```
▼ Scene
Camera
  Perspective
Clear
R:204   G:204   B:204   A:255   Color
Directional Light
R:108   G:107   B:107   Ambient
R:204   G:204   B:204   Diffuse
R:255   G:255   B:255   Specular
0.000   0.707   0.707   Direction
```

While researching PBR, preliminary work was completed on the renderer. The intention was to have the renderer complete enough so that upon completing this report, efforts could be focused on producing a working PBR solution.



```
▼ Scene
Camera
  Perspective
Clear
R:204   G:204   B:204   A:255   Color
Directional Light
R: 51   G: 51   B: 51   Ambient
R:204   G:204   B:204   Diffuse
R:255   G:255   B:255   Specular
0.000   0.707   0.707   Direction
```

The current version of the project allows PBR glTF 2.0 models to be imported and rendered. The mouse can be used to rotate and magnify currently imported models. The scene is configurable through a basic user interface. The shading model employed is a Blinn-Phong model. A basic threading model is implemented allowing the main thread to send draw commands asynchronously to a rendering worker thread. The threads must synchronise when starting a new frame, executing all commands currently enqueued. Sort-based draw call bucketing is implemented, minimising OpenGL state changes and allowing order-dependent effects to be submitted properly.

The systems described in section 5.4 have been implemented, allowing the remainder of the project to focus on creating PBR shaders.

This code snippet shows how a model is currently rendered in Moka:

```
void draw(const game_time delta_time) override
{
    command_list scene_draw;

    scene_draw.viewport()
        .set_rectangle(0, 0, 1280, 720);

    scene_draw.scissor()
        .set_rectangle(0, 0, 1280, 720);

    scene_draw.clear()
        .set_color(clear_color_)
        .set_clear_color(true)
        .set_clear_depth(true);

    // iterate over all meshes in a model
    for (auto& mesh : model_)
    {
        // iterate over all the primitives in the mesh
        for (auto& primitive : mesh)
        {
            // set all the material properties necessary to render
            auto& material = primitive.get_material();

            const auto distance = glm::distance(
                mesh.get_position(), camera_.get_position());

            material["model"]      = mesh.get_transform().to_matrix();
            material["view"]       = camera_.get_view();
            material["projection"] = camera_.get_projection();
            material["view_pos"]   = camera_.get_position();

            material["light.ambient"]  = light.ambient;
            material["light.position"] = light.position;
            material["light.diffuse"]  = light.diffuse;
            material["light.specular"] = light.specular;
            // generate a sort key for this command buffer
            // based on the distance from the camera & material data
            const auto key = sort_key_generator_(
                distance,
                material.get_program().id,
                material.get_alpha_mode());

            // make a new command buffer, give it the sort key
            auto& command_buffer = scene_draw.make_command_buffer(key);

            // populate the new command buffer
            primitive.draw(command_buffer);
        }
    }

    // submit the command list and swap the buffer
    graphics_.submit_and_swap(std::move(scene_draw));
}
```

8. Future Work

Decent progress has been made with the Moka implementation. However, some areas of the codebase need to be revisited - a substantial number of the classes are defined fully in the header files, and many are not explicitly referenced in CMake. The solution will need to be tidied and organized before any work can continue to prevent the code quality degenerating any further. The future development can be made much easier by devoting a few days to improving the structure and organisation of the solution.

Version control use needs to be improved moving forward. Commits have been massive and so far branches have not been merged back into their parents. Committing early and often will prevent disaster from loss of work.

The next step is implementing a physically based BRDF in the fragment shader. All the necessary work is in place for this, it should only take altering the existing shaders before an improvement can be seen in how plausible materials look. Once this is completed, the focus will be shifted to completing diffuse light probes and creating irradiance environment maps on the GPU. The specific method that will be used is not yet known. Many examples exist for a simple importance sampling solution, however, the spherical harmonics method is widely used so more research must be conducted before committing to an implementation. To get diffuse light probes working, the renderer must be extended to allow for framebuffers and cube maps. Once diffuse probes are completed, specular light probes will be tackled, likely using Karis' (2013) method.

A presentation is to be held to discuss the contents of the research. This project's presentation will take inspiration from the content of PBR presentations held at SIGGRAPH.

The survey must be completed and distributed to evaluate the effectiveness of the solution. This puts an unfortunate time strain on the project - the renderer must be complete and ready to render PBR images before this survey can be distributed.

The final honours report will expand upon many of the areas covered in this report. Much more focus will be given to the implementation produced, and some of the areas that are given a high-level overview in this report, such as light probes, will be given a much more in-depth discussion.

10. 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.

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].

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].

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].

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

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.

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].

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].

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].

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.

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

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. (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.

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: Advanced Rendering Techniques*. 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.

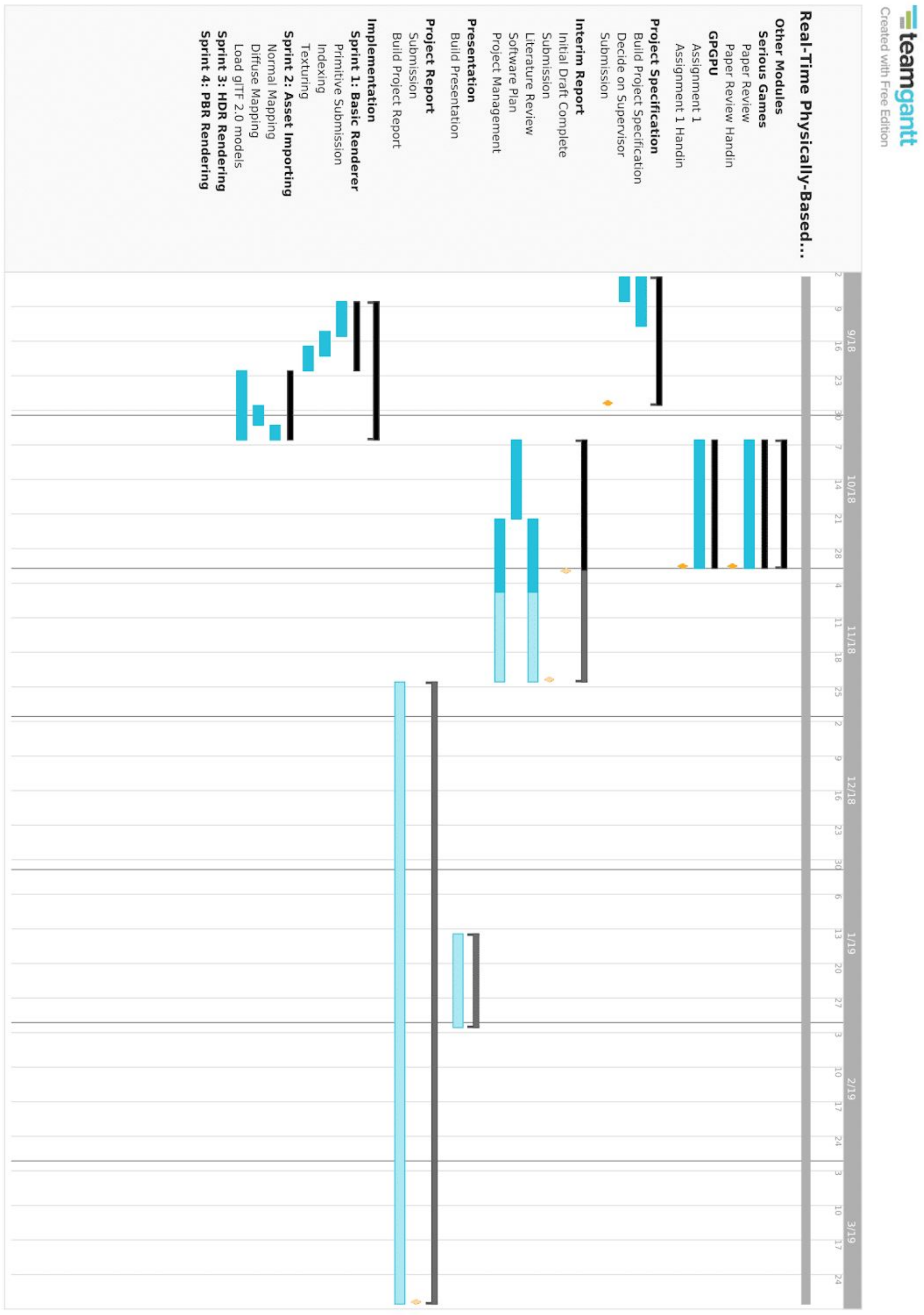
Trowbridge, T. and Reitz, K. (1975). *Average Irregularity Representation of a Rough Surface for Ray Reflection*. *Journal of the Optical Society of America*, 65(5), pp.531-536.

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.

11. Appendices

11.1. Gantt Chart



11.2. 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