School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

Final Report

Physically Based Shading Models in Real-time Rendering

Evan James

Submitted in accordance with the requirements for the degree of
Computer Science (Digital & Technology Solutions) BSc

2021/22

COMP3932 Synoptic Project

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	Uploaded to Minerva (DD/MM/YY)
<Example> Scanned participant consent forms	PDF file / file archive	Uploaded to Minerva (DD/MM/YY)
<Example> Link to online code repository	URL	Sent to supervisor and assessor (DD/MM/YY)
<Example> User manuals	PDF file	Sent to client and supervisor (DD/MM/YY)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) _____

Summary

<Concise statement of the problem you intended to solve and main achievements (no more than one A4 page)>

Acknowledgements

<The page should contain any acknowledgements to those who have assisted with your work. Where you have worked as part of a team, you should, where appropriate, reference to any contribution made by other to the project.>

Note that it is not acceptable to solicit assistance on 'proof reading' which is defined as the "the systematic checking and identification of errors in spelling, punctuation, grammar and sentence construction, formatting and layout in the test"; see
`https://www.leeds.ac.uk/secretariat/documents/proof_reading_policy.pdf`

Contents

Chapter 1

Introduction and Background Research

1.1 Introduction

Rendering is the process of generating images, or *frames*, of a virtual world (known as a *scene* in rendering). Real-time rendering requires that the generation of these frames is done at a fast enough rate so that the viewer feels they are taking part in an immersive, dynamic experience. Typically, this rate needs to be at least 30 FPS (Frames Per Second), with 60 FPS and beyond being desirable [1]. This imposes a maximum time budget of 33 to 16 milliseconds in which each frame must be generated, the *frame time*. Real-time rendering presents a compelling problem: how can the visual fidelity of a rendered scene be maximised, whilst adhering to this strict computational budget.

Rendering can be performed using one of two techniques, ray tracing or rasterization. Ray tracing is based on a model that is analogous to how humans perceive light and colour in the real-world. In the real-world, rays of light are produced from many sources, bounce from one object to the next, and eventually reach the viewers eyes. Ray tracing models this same process, but in reverse, with the rays originating from the views eyes, and being traced back to their sources. Provided enough rays are sampled, this approach produces very realistic images. Although ray tracing is the standard in the realm of movie production, its expensive computational requirements lead to frame times in the region of minutes instead of milliseconds [2]. Aside from so notable exceptions[1], this prohibits its use in real-time applications. As a result, real-time rendering employs another technique, rasterization.

With rasterization, each object in the world is composed of an arrangement of primitive shapes, most commonly, triangles, and their material is described through a number of parameters. When rendering, the world is transformed and projected onto a 2D plane. Within this plane, a fixed region maps to the space of the output image; all triangles that lie outside this region are clipped. The remaining triangles are then split into granular pieces, called *fragments*. A colour is calculated for each fragment by evaluating the amount of light that shines on that fragment in the world, and then how that light interacts with the material of the object that fragment belongs to. Performing this calculation is called *shading*, and how it is done is defined by a *shading model*. After resolving which fragments lie on top of which others, the final image is presented to the user. This whole rasterization process is referred to as the graphics rendering pipeline, and dedicated hardware has been developed to carry it out, the *Graphics Processing Unit* (GPU).

The appearance of the final rendered frames is largely determined by the shading model, and therefore the choice of such a model is crucial. For a long time, the standard shading model used for photo-realistic real-time rendering was Blinn-Phong; it was utilised in popular game engines, and was the default model used in OpenGL's fixed function pipeline [5] [6] [7]. Blinn-Phong is an empirical model: it is based on human observations of how light interacts with materials, rather

[1] With the introduction of hardware accelerated ray tracing on consumer GPUs [3], the use of ray tracing to render specific visual phenomena, such as reflections, has seen use in some modern games [4].

than the underlying real-world physical rules that govern those interactions [8]. Blinn-Phong can produce reasonably realistic images, and is computationally inexpensive - a very desirable trait for real-time rendering. However, due to its non-physically based nature, Blinn-Phong has many issues. Paramount amongst which is its inability to render certain physical phenomena, which limits the realism of rendered frames. Furthermore, the parameters of Blinn-Phong that are used to specify material properties, bear little relation to the characteristics of physical materials. This problem manifests itself in a tight coupling between material parameters and lighting conditions. In order to accurately depict the same physical material under different lighting conditions, it may be necessary to specify differing values for these parameters. This reduces the reusability of assets, making artist workflow more difficult.

In an effort to alleviate these issues, the replacement of Blinn-Phong in favour of physically based shading models has seen widespread adoption. Such models work by evaluating equations that simulate the real world physical interaction of light and objects. Using these models for shading is known as *Physically Based Shading* (PBS), and their use in the wider rendering pipeline is called *Physically Based Rendering* (PBR). PBS represented a seismic shift in the real-time rendering industry, with major game engines migrating to a PBR pipeline [9] [10].

The aim of this project is to investigate the use of physically based shading models in real time rendering. Specifically, I will seek to highlight the benefits of PBS when compared to the technology is superseded, Blinn-Phong shading.

The advantages of using PBS over Blinn-Phong shading can be broadly categorised into two groups: the improvements to artist workflow; and the improved photorealism. As mentioned previously, because of how materials are defined in Blinn-Phong shading they are often not portable between different lighting environments. In contrast, the parameters that determine materials in PBS are based on physical properties. This permits the reuse of materials and assets over different lighting configurations [10] [11]. Burley outlines how this reduction in the need for "material 're-do's" yields an extremely significant improvement to artist workflow [12]. Although these benefits are an important motivating factor for using PBS, the practical issues that arise from trying to investigate and quantify them (I don't have access to a team of artists) mean that this report will focus solely on exploring those advantages in the latter category – how does PBS render frames that are more photorealistic than Blinn-Phong?

Answering this question by simply commenting on the general perceived realism of a frame when compared to another is a largely subjective exercise. Instead, in a concerted effort to be as objective as possible, I will examine the benefits of PBS by identifying physical phenomena that are more accurately modelled in frames rendered using PBS, than in frames rendered using Blinn-Phong shading. To this end, I will be developing a renderer that can render scenes using both Blinn-Phong shading, and PBS.

1.2 Background Research

After conducting considerable research within the area of real-time shading models, it is evident that a comparison of the nature described above, has not been done before. Therefore, no descriptions of previous comparisons can be given. Instead, the research presented below thoroughly explores the relevant literature and theory behind Blinn-Phong shading and PBS. This serves

two purposes. First, it forms the knowledge that is necessary to carry out the implementation of the renderer. Secondly, it allows the later comparison between the two shading approaches to be performed, and crucially, the results of that comparison to be interpreted and justified.

The research begins with a discussion of the Blinn-Phong shading model. We then delve into the physics of how light interacts with matter, and how this pertains to shading. An exploration of the theory underpinning physically based shading models follows, and then we present several such models. After, we consider how lights can be represented in a physical manner. Finally, we finish by focusing on the wider PBR aspect with a discussion on the nature of shaded pixel values, and the transformations that need to be applied before passing those values to the display. The mathematical notation used throughout this report is given in Appendix C.

1.2.1 Blinn-Phong Shading

Phong Model

In 1975, Phong introduced a simple shading model for rendering realistic images [8]. The original model is parametrised as the sum of two terms, *diffuse* and *specular*, but in practice it is commonly supplemented with a third term, *ambient*. Each one describes the contribution of a different lighting component. Splitting shading into the evaluation of a diffuse and specular term is common practice, and the physical basis for doing this is explained in section 1.2.2.

An ideal diffuse surface is one that has a Lambertian response to incident light, where the light is reflected in all directions equally [13]. Therefore, the determining factor in the appearance of such surfaces is the intensity of the incident light, which is a function of the direction of incident light and the orientation of the shaded surface. This is called *Lambert's Cosine Law* [13]. The diffuse term encodes the lighting effects of parity between a primitive's surface orientation and the direction of the light, with surfaces facing the light being illuminated more intensely than those facing away from it. This behaviour is formulated as:

$$\mathbf{c}_{shaded_{diff}} = \mathbf{c}_{surface_{diff}} \mathbf{c}_{light_{diff}} (\mathbf{n} \cdot \mathbf{l})^{+} \tag{1.1}$$

$\mathbf{c}_{shaded_{diff}}$, $\mathbf{c}_{surface_{diff}}$, and $\mathbf{c}_{light_{diff}}$ are the RGB triplets that represent the diffuse colour of the shaded fragment, the diffuse colour of the surface, and the diffuse colour of the incident light respectively. This separation of the light and surface colours into separate components was not present in Phong's original model. However, many implementations have increased the flexibility of the model by exposing these additional parameters [14]. The *normal*, $\mathbf{n}$, is the unit vector pointing away from the surface at the shaded point, giving the orientation of the surface. The *light direction*, $\mathbf{v}$, is the unit vector pointing in the direction of the incident light. See Figure 1.1 for an illustration of the principle vectors used in the Phong shading model (and indeed, by most shading models). The dot product of two unit vectors is equivalent to taking the cosine of the angle between them. Therefore, $(\mathbf{n} \cdot \mathbf{l})^{+}$ will increase from 0 to 1 as the angle between the incident light direction and the surface normal decreases. Thus, the more aligned the surface orientation and light direction are, the greater the intensity of the diffuse term. Negative values of the dot product indicate that the light direction is underneath the surface. In these cases the light is not incident upon the surface at all, so the dot product is clamped to 0.

The specular term of the model captures the ability for surfaces to exhibit highlights due to

surface reflections. When light is incident upon a surface, it will experience some reflection, and when the reflected light is aligned with the direction of the viewer, this is perceived as a region of increased illumination, a *specular highlight*. The formula for the specular term is:

$$\mathbf{c}_{shaded_{spec}} = \mathbf{c}_{surface_{spec}} \mathbf{c}_{light_{spec}} ((\mathbf{r} \cdot \mathbf{v})^{+})^{surface_{shininess}} \tag{1.2}$$

Where $\mathbf{r}$ is the reflection of the incident light about the surface normal, and is defined as:

$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{l})\mathbf{n} - \mathbf{l} \tag{1.3}$$

The RGB triplets are similar to those in the diffuse equation, except these are specific to the specular response. Phong's original model had the colour of the specular highlight be the same as the overall colour of the light (not split into separate diffuse and specular components). This gave all materials an overly plastic appearance. Introducing the $\mathbf{c}_{surface_{spec}}$ and $\mathbf{c}_{light_{spec}}$ variables allows for the colour of the specular highlight to be fully configurable, mitigating this issue [14]. The *view vector*, $\mathbf{v}$, is the unit vector pointing in the direction of the viewer. The dot product measures the alignment between the view direction, and the direction of the reflected incident light. The $surface_{shininess}$ parameter determines the concentration of the reflected light rays. The higher the value, the more focused the reflected rays are, the smaller the specular highlight becomes, and the shinier the object appears. Typical values range from 1 to 100.

Finally, we have the ambient term. In the model developed so far, if a shaded point is not directly visible from a light source, then it will be black. In reality, such points are never completely unilluminated - rays from light sources will bounce around the environment, eventually lighting these obscured areas. So far we have only considered *direct illumination*; the ambient term is used to crudely approximate the lighting that is a consequence of this *indirect illumination*:

$$\mathbf{c}_{shaded_{ambi}} = \mathbf{c}_{surface_{ambi}} x \tag{1.4}$$

$\mathbf{c}_{surface_{ambi}}$ controls the colour of the ambient shading. Typically, it is just set equal to $\mathbf{c}_{surface_{diff}}$. x is a constant value defined for the whole scene, rather than per light, and controls the amount of indirect illumination that occurs. Small values of x should be used, as anything too large will make the scene look unrealistically bright - lighting via indirect illumination is usually quite subtle as light loses energy every time it reflects of a surface.

These three terms are summed together to give the overall Phong shading model:

$$\mathbf{c}_{shaded} = \mathbf{c}_{shaded_{ambi}} + \mathbf{c}_{shaded_{diff}} + \mathbf{c}_{shaded_{spec}} \tag{1.5}$$

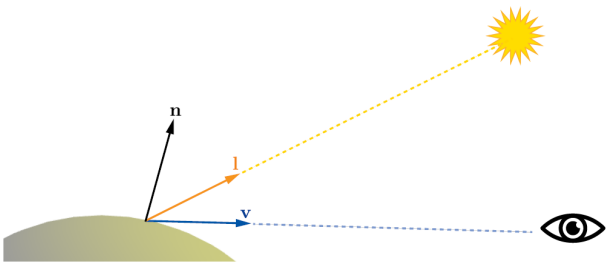


Figure 1.1: The surface normal $\mathbf{n}$, light direction $\mathbf{l}$, and view vector $\mathbf{v}$. Taken from [15]

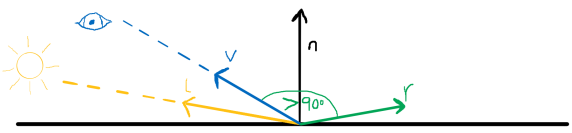
Blinn-Phong Model

One of the issues with the Phong shading model is made apparent when viewing a rough (low *shininess* value) surface from a direction close to the incident light. The corresponding reflected light vector makes an angle with the view direction that is greater than 90°. In this instance, the dot product $\mathbf{r} \cdot \mathbf{v}$ evaluates to a negative value, and is thus clamped to 0, leading to no specular contribution. However, for very rough surfaces, the specular highlight is so wide that even at these greater angles, there should still be a specular contribution. See Figure 1.2a for an illustration of the problem.

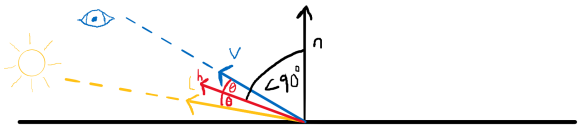
In 1977, Blinn remedied this issue by modifying the Phong shading model with a more accurate specular term [16]. He utilised the half vector, $\mathbf{h}$, dispensing with the reflected light vector, $\mathbf{r}$, and replaced the existing specular dot product with $\mathbf{h} \cdot \mathbf{n}$. $\mathbf{h}$ is a unit vector pointing in the direction that is halfway between the $\mathbf{l}$ and $\mathbf{v}$ vectors. It is calculated as:

$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|} \tag{1.6}$$

Blinn's specular term emulates the overall behaviour of the original Phong term. As the (now conceptual) reflection vector $\mathbf{r}$ aligns with $\mathbf{v}$, so does the half vector $\mathbf{h}$ align with the surface normal $\mathbf{n}$. Crucially though, $\mathbf{h} \cdot \mathbf{n}$ will only evaluate to a negative value, and subsequently be clamped to 0, if $\mathbf{l}$ or $\mathbf{v}$ is beneath the surface. Therefore, the scenario in which rough surfaces were being shaded incorrectly with no specular contribution, is resolved. See 1.2b for an illustration.



(a) The angle between $\mathbf{r}$ and $\mathbf{v}$ can exceed 90° whilst $\mathbf{l}$ and $\mathbf{v}$ are still above the surface



(b) The angle between $\mathbf{h}$ and $\mathbf{n}$ can't exceed 90° without $\mathbf{l}$ or $\mathbf{v}$ being below the surface

Figure 1.2

Blinn's modification to the Phong model is known as the Blinn-Phong shading model, and it produces more realistic images than the original. The model is formulated as:

$$\mathbf{c}_{shaded} = p_{ambi} + p_{diff}(\mathbf{n} \cdot \mathbf{l})^{+} + p_{spec}((\mathbf{h} \cdot \mathbf{n})^{+})^{surface_{shininess}} \tag{1.7}$$

where

$$\begin{aligned} p_{ambi} &= \mathbf{c}_{surface_{ambi}} x \\ p_{diff} &= \mathbf{c}_{surface_{diff}} \mathbf{c}_{light_{diff}} \\ p_{spec} &= \mathbf{c}_{surface_{spec}} \mathbf{c}_{light_{spec}} \end{aligned}$$

Expressing the model in this way draws attention to the relative proportions of each component that contributes to the overall shading. These proportions, p, are modulated via lighting parameters: $light$ variables and x; and material parameters: $surface$ variables.

In scenes with multiple lights, the diffuse and specular terms of equation 1.7 are evaluated for each one. Then these intermediate values are summed together with the scene's ambient term to get the overall colour of the fragment.

1.2.2 Physics of Light-Matter Interaction

Visible Light

The Electromagnetic spectrum is a distribution of wavelengths, of which a small section is spanned by visible light. More specifically, visible light are those electromagnetic (EM) waves with a wavelength between 400nm, violet, and 700nm, red. Typically, visible light will contain a combination of wavelengths within this range. We can express light waves as a relation between each wavelength and its associated energy, in what are known as Spectral Power Distributions (SPD). Although s exist that store light as SPDs, the negative performance implications that accompany this approach mean that in practice, an abstraction is often used [17]. This abstraction exploits the relative imprecision of the human visual perception system. Using a linear combination of three wavelengths, R, G and B, we can accurately represent any visible light wave and colour [18]. Renderers store this combination as an RGB triplet, with each value giving the weight of its associated wavelength.

The area of study concerned with quantifying EM waves is called *Radiometry* [19]. Several *radiometric* measurements exist, with the most commonly used one in rendering being *radiance*, L. Radiance measures the power of a single EM ray. It's defined as energy over time (power), with respect to area and direction: W/m^2sr. Power is expressed in Watts, area in metres squared, and direction in *Steradians*. Steradians are to solid angles, what radians are to normal angles. A solid angle is the 3D version of a normal 2D angle. Evaluating a shading model is equivalent to computing the radiance of a single light ray that goes from the shaded fragment to the viewer. See section 1.2.3 for more details.

Light Interaction with Matter

The way in which light interacts with matter is characterised by the matter's *index of refraction* (IOR). The IOR defines the speed that light travels through the matter, given in relative terms to the speed of light in a vacuum [20]. The IOR can also be extended to describe the absorption qualities of the matter, in what is known as the *complex index of refraction*. Some matter absorbs light waves by converting part of their energy to heat, causing an attenuation of the lights amplitude and intensity. Both forms of refractive index can vary by wavelength.

Light interacts with matter in three ways [21]. When light is travelling through a volume of matter with a uniform IOR, a *homogenous medium*, it will not deviate in its direction. However, the absorption value of the IOR may cause the light to undergo changes in intensity, and if this varies by wavelength, changes in colour also. Water is an example of a homogenous medium that absorbs light, but mostly around the red wavelengths - this gives its intrinsic blue and green colour [20].

Light behaves differently in a *heterogeneous medium*, where the IOR is not uniform. If light encounters a change in IOR that takes place over a distance smaller that a light wavelength, it will *scatter* into multiple directions. The amount of scattering varies by wavelength. The frequency of scattered light will be the same as the original light. The only exception to this is when fluorescence or phosphorescence phenomena is present, but since this is very rare in the real world, we ignore such scenarios when rendering. In the likely case that the original light is comprised of a distribution of different wavelengths and frequencies, each one will interact

independently. If interacting with a single isolated particle, the scattered light will move in all directions, but at different intensities [15].

Finally, light can interact with matter in a manner that is opposite to that of absorption. *Emission* can take place within matter, where other forms of energy are converted to light energy, and light is emitted. Light sources work in this way.

In general, the appearance of most media is as a consequence of both scattering and absorption.

Light Incident to Surfaces

When shading, we are interested in simulating what happens when light is incident upon the surface of an object. When this interaction occurs, two properties effect the outcome: the geometry of the surface, and the nature of the substances either side of the surface [15] [22].

We first consider the substance factor and assume that the surface's geometry is a perfectly flat plane. An object's surface can then be described as a flat two-dimensional boundary separating two volumes with differing IORs. In rendering it is typical that the outer volume is comprised of air, which has an IOR of 1 (or to be exact, 1.0003). The inner volume has an IOR defined by the material of the object. Any light wave that impinges on the boundary will encounter an abrupt change in the IOR, which - as explained above - will result in scattering. Because the boundary is a flat plane, the nature of this scattering is well defined: some of the scattered waves will continue moving into the surface, the *transmitted wave*, and the others will be reflected at the surface and move away, the *reflected wave*. The reflected wave propagates in a direction that is the reflection of the incident wave about the surface normal. The transmitted wave will undergo *refraction* and move at an angle of θ_t, which is defined by the relative IOR of the two volumes and the angle of the incident light, θ_i [23]. See Figure 1.3 for an illustration of this scattering behaviour. The proportion of reflected versus transmitted light follows the Fresnel equations, which are explained in section 1.2.4 [24]. The transmitted wave will interact with the interior of the object in the same way as light interacts with any medium - it will experience some degree of scattering and absorption.

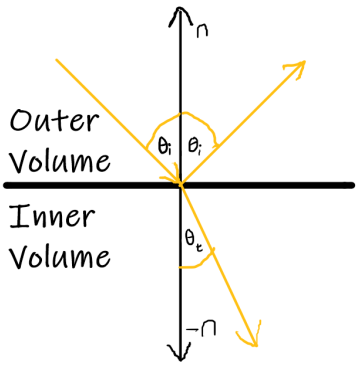


Figure 1.3: The scattering that occurs when light is incident upon a perfectly flat boundary

Now we focus on the effect that geometry plays when light is incident on a surface. The case we have just seen of a perfectly flat surface is of course not possible; all surfaces have some variation in their geometry. Any irregularities in the geometry that are smaller than a single light wavelength do not have any impact. Irregularities larger than 100 wavelengths constitute their own local plane of flatness, with their own orientation. Irregularities that have a size within

this range cause the surface-light interaction to behave differently to that described above, with phenomena such as *diffraction* also being present. However, when shading we typically stay within the realm of *geometrical optics*, which means we ignore the effects of these irregularities[2] [22]. Geometrical optics strictly deals with light as rays, not waves, and these rays always intersect locally flat planes with behaviour described above and illustrated in Figure 1.3. When shading, a single pixel or fragment will span much more than 100 wavelengths, so we need to account for the local planes of flatness that exist in this region. Irregularities at this scale are referred to as *microgeometry*. As a consequence of microgeometry, each point on the surface will reflect light in one direction, and refract it in another. Therefore, when determining the effect of microgeometry over a whole pixel, we can consider the light to be reflected and refracted in many directions. These directions are defined by the individual orientations of the points, and quantified by regarding them as a distribution of normals. The tighter the distribution, the tighter the spread of reflected and refracted directions. The roughness of the material directly controls the variance of this distribution. Section 1.2.4 discusses this in detail.

Transmitted Light Interaction

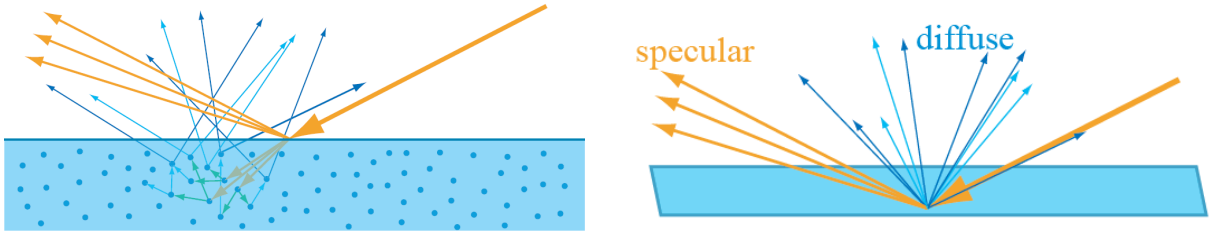
As explained previously, the light that is transmitted into the interior of an object will experience a mixture of scattering and absorption. In some materials, the light will be scattered enough that it is re-emitted at the surface of the object in a process called *subsurface scattering* [26]. See Figure 1.4a for an illustration. The distance between the subsurface scattered light and the original incident light is determined by the nature of the material, and is very important when shading. If the subsurface scattering distances are smaller than the span of one pixel, as is common for most materials, we call it *local subsurface scattering* [24]. In such cases, the subsurface scattered light is combined with the light reflected from the object surface to form a local shading model, where the outgoing light at one point is wholly dependent on the incoming light at that same point. Subsurface scattered light will encounter protracted interactions of absorption and scattering in the interior of the object, before finally being re-emitted. Therefore, it will have a distinctly different colour to the surface reflected light [27]. For this reason, shading is split into two different terms: the *specular term* captures the light reflected at the object surface, whilst the *diffuse term* models the local subsurface scattering. See Figure 1.4b for an illustration of these two shading components.

Shading is more complex when the subsurface scattering distances are larger than a single pixel. *Global subsurface scattering* is often encountered when rendering skin or wax, and although special models have been developed to shade these materials, it is beyond the scope of this report [28].

Metals and Dielectrics

The proportion of scattering and absorption that transmitted light induces varies by material. There are two main categories of materials that we encounter day-to-day: metals and *dielectrics*. Metals immediately absorb any transmitted light, meaning their appearance is solely provided by the specular term. Most other materials are non-conductive, called dielectrics. These materials

[2]Although not widely used in real-time rendering, some shading models do exist for simulating the phenomena that occur when light interacts with geometrical irregularities of this size [25]



(a) Light is re-emitted at the surface of an object by means of subsurface scattering

(b) When shading a point we consider the specular and diffuse terms

Figure 1.4: Taken from [15]

do allow for subsurface scattering, so both the specular and diffuse terms characterise their appearance.

1.2.3 Principles of Shading

The Reflectance Equation

When rendering, we model the viewer as a camera placed at point $\mathbf{c}$. The camera projects a 2D matrix of photosensitive sensors that map to pixels on the display. By combining outputs of all the sensors, we get the final rendered image. For each of the sensors, we have a ray that originates from $\mathbf{c}$, intersects the sensor, and then continues into the scene; this ray propagates in the opposite direction to the view vector, $-\mathbf{v}$. With PBS, we obtain the colour of a sensor by calculating the incoming radiance to $\mathbf{c}$ in the direction $-\mathbf{v}$. Thus, the final rendered image is given by calculating incoming radiance to $\mathbf{c}$ over the set of all $-\mathbf{v}$ vectors.

A scene is comprised of a number of objects separated by media. Typically when rendering, we just consider all media to be air. Air is a medium that exhibits very little scattering nor absorption, so its effect on radiance is minimal and can be ignored. Therefore, incoming radiance to $\mathbf{c}$ in the direction $-\mathbf{v}$, is equivalent to outgoing radiance from point $\mathbf{p}$ along $\mathbf{v}$, where $\mathbf{p}$ is the intersection point that the closest object will make with a ray that travels along $-\mathbf{v}$. This quantity is defined as $L_o(\mathbf{p}, \mathbf{v})$ and is calculated by means of the *reflectance equation*:

$$L_o(\mathbf{p}, \mathbf{v}) = \int_{\mathbf{l} \in \Omega} f(\mathbf{l}, \mathbf{v}) L_i(\mathbf{p}, \mathbf{l})(\mathbf{n} \cdot \mathbf{l})\, d\mathbf{l} \tag{1.8}$$

The $\mathbf{l} \in \Omega$ in the integral subscript says that the integral should be performed over all directions $\mathbf{l}$ that exist in the unit hemisphere Ω, which is centred on $\mathbf{p}$ and orientated along the surface normal $\mathbf{n}$. In this way, the incoming radiance from all possible light sources is considered. The product within the integral gives the outgoing radiance from $\mathbf{p}$ along $\mathbf{v}$ for a singular incident light direction $\mathbf{l}$. The effect of the integration is to sum over all the individual components of outgoing radiance, to obtain the total value. This formulation is consistent with the shading scenario that was introduced in the "Transmitted Light Interaction" section of 1.2.2, except it is expressed from a slightly different perspective: instead of keeping the incoming light direction constant and considering many outgoing light directions, we are now keeping the outgoing light direction constant, and considering all possible incoming light directions. Note that this is indeed a local shading model, with all outgoing light at $\mathbf{p}$ being wholly dependent on incoming light at $\mathbf{p}$. A physically based shading model computes the reflectance equation using physically based

implementations of f and L_i. We now discuss each of the factors of the integral's product in turn.

The BRDF

The *bidirectional reflectance distribution function* (BRDF), $f(\mathbf{l}, \mathbf{v})$, was first introduced by Nicodemus et al in 1977 [29]. For a given $\mathbf{l}$ and $\mathbf{v}$, the BRDF gives the ratio of incident light in direction $\mathbf{l}$, which after striking the surface, is reflected in direction $\mathbf{v}$. The function is a distribution over all possible values of $\mathbf{l}$ and $\mathbf{v}$ that lie in the unit hemisphere introduced above, and thereby it completely describes a surface's local reflectance response to incident light. Local reflectance encompasses both surface reflection and local subsurface scattering. Variations of the BRDF exist which seek to capture other light-matter interactions, such as the BSSRDF which accounts for the influence of global subsurface scattering [30].

As explained previously, scattering and absorption - the two phenomena that underpin local reflectance - vary by wavelength. Therefore, the BRDF also needs to vary by wavelength, so the ratios it returns are given as RGB triplets.

For a BRDF to be considered physically plausible, two constraints must hold. The first is called the Helmholtz reciprocity and states that $f(\mathbf{l}, \mathbf{v}) = f(\mathbf{v}, \mathbf{l})$ [31]. BRDFs used in rendering don't often comply with this equality, but still look physically correct [15]. The second constraint is imposed by conservation of energy: the outgoing light energy cannot exceed the incoming light energy. In real-time rendering, exact adherence to this principle is not required, but respecting it in the approximate sense is very important - otherwise, objects will be rendered as overly bright and unrealistic [15][10].

Constructing a BRDF can be accomplished in two ways: via optical measurements of real materials, or by leveraging mathematical formulas. Ward discusses the use of a *Gonioreflectometer* to measure BRDFs of real world materials [32]. This process is time consuming and thus it's use in rendering is impractical. Instead, we construct BRDFs using parametrised mathematical formulas. The parameters involved are properties of the object's material. If the material properties are specified over every point on the object - which is commonly practiced in graphics by utilising textures - then this is equivalent to defining a BRDF for every point on the object's surface. As mentioned, the local reflectance response of a surface is split into specular and diffuse terms when shading. Naturally, this extends to the BRDF itself, and it is expressed as the sum of a specular BRDF and a diffuse BRDF.

These parametrised mathematical formulas can be categorised as either empirical or physically based. Although not represented explicitly, the Blinn-Phong shading model detailed in section 1.2.1 defines within it an empirical BRDF. Physically based BRDFs lie at the very heart of PBS, and sections 1.2.4, 1.2.5 and 1.2.6 are concerned with investigating them.

Figure 1.5 gives an example of a BRDF. Visualising a BRDF is difficult as it is a function of four parameters (two angles per vector, one for elevation and another for rotation), so the Figure adopts the common approach of keeping the incident light direction constant.

Incoming Radiance

The incoming radiance term, $L_i(\mathbf{p}, \mathbf{l})$, represents the light that originates in direction $\mathbf{l}$ and strikes the surface at point $\mathbf{p}$. Illumination and light sources are discussed in depth in section 1.2.7.

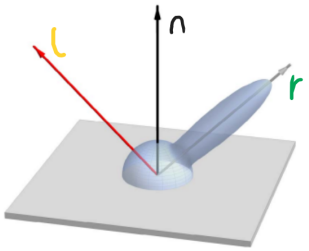


Figure 1.5: An example BRDF. The spherical shape represents the diffuse response, and the lobe represents the specular response. Notice that the specular lobe is concentrated around the reflection vector **r**. Adapted from [33].

Dot Product Term

The dot product term, $(\mathbf{n} \cdot \mathbf{l})$, is the same as that discussed in section 1.2.1. It fulfils the same purpose, going from 0 to 1 as the light direction approaches the surface normal. When multiplied by the incoming radiance term, it appropriately scales the amount of illumination. It's also common for this dot product to be clamped to 0, so that contributions from light sources below the surface are ignored.

1.2.4 BRDF Building Blocks

Most physically based BRDFs are built upon the same set of standard mathematical functions and theory. The individual BRDFs then differ from one another in how these functions are implemented and compiled together. We first detail the standard mathematical building blocks, and then specific specular and diffuse BRDFs are discussed in sections 1.2.5 and 1.2.6 respectively.

Fresnel Reflectance

In section 1.2.2, the behaviour of light that impinges upon a surface was said to be dependent on two factors: the substances either side of the surface, and the geometry of the surface. As alluded to, the Fresnel equations describe how light behaves due to the substances. We again assume a perfectly flat boundary separating an outer and inner substance, which have IORs of n_1 and n_2 respectively. The Fresnel reflectance, F, gives the proportion of incoming light that is reflected at the boundary. Due to the conservation of energy, the amount of transmitted light can then also be easily obtained. F varies by wavelength so is expressed as an RGB triplet. Given values of n_1 and n_2, F is then a function of incident light angle, $F(\theta_i)$. As mentioned, in rendering it's typical for the outer substance to be air with an IOR of 1, and thus we will be in a scenario of *external reflection*, where $n_1 < n_2$.

In the case of external reflectance, $F(\theta_i)$ always follows the same pattern [15]. When $\theta_i = 0°$, F will be equal to the intrinsic specular colour of the inner substance, F_0. Between 0° and 90°, F will then increase non-linearly towards white; slowly for most of the interval and then rising rapidly when close to 90°. The tendency for surfaces to exhibit increased reflection at glancing angles is called the *Fresnel effect*.

Using the Fresnel equations themselves is not possible in real-time rendering because n_1 and n_2 must be known for all wavelengths of visible light, and this data is not available [34]. Instead, approximations that rely on the pattern outlined above are utilised. These are functions that

depend in part on F_0, which is known for many materials.

F_0 is defined by the object's material parameters. Dielectrics all have very low F_0 values, grouping around the $(0.04, 0.04, 0.04)$ mark - a fact that is exploited by many physically based shading models. This means dielectrics only reflect a noticeable amount of light at very glancing angles. Metals on the other hand have much higher F_0 values, with each RGB channel typically exceeding 0.5. Akenine-Möller et al give a comprehensive overview of F_0 values for different materials [15].

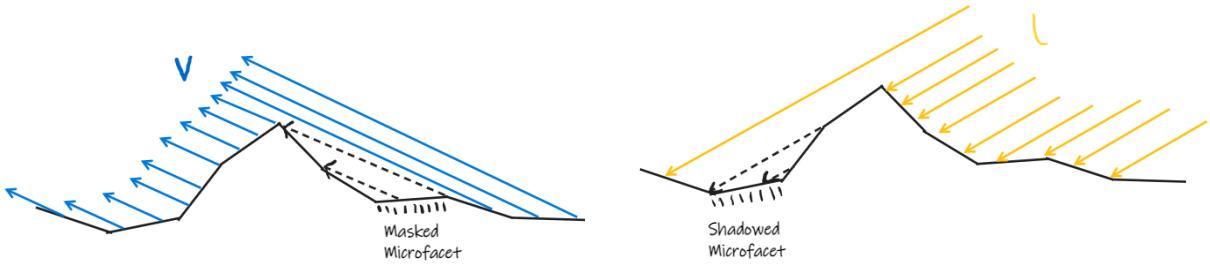
Microfacet Theory

We now revisit the role that the surface geometry plays in local reflectance. Recall that when rendering, a single pixel will lie over many pieces of microgeometry. The reflectance at the shading point is then determined by the aggregation of the individual interactions that light will have with each piece of microgeometry. We reason about these interactions by utilising *microfacet theory*. The theory states that microgeometry be modelled as a collection of perfectly flat planes called *microfacets* [16]. Such a model is effective at representing many real world materials [35]. The microfacets' impact on surface reflectance is described by a *Normal Distribution Function* (NDF), a *masking-shadowing function*, and a *micro-BRDF*.

The NDF, $D(\mathbf{m})$, statistically describes the orientations of the microfacets over the macro-surface (the shaded point when viewed at the pixel scale) [36]. The more microfacets that have a normal of $\mathbf{m}$, the higher the value of $D(\mathbf{m})$. Typically, surfaces posses a distribution of microfacet normals that is concentrated around the normal of the macrosurface $\mathbf{n}$.

Whilst the NDF describes the orientations of the microfacets, it doesn't describe their arrangement, which also plays a significant role in how light reflects off the macrosurface. Some microfacets will not be visible from the view vector $\mathbf{v}$ because they are occluded by other microfacets. This is known as *masking*. Only those microfacets that are not masked by others will be visible to the viewer and thus contribute to the shading. Furthermore, some microfacets will be occluded so they aren't visible to the incoming light direction $\mathbf{l}$. This is known as *shadowing*. Microfacet theory only models the first interaction between light and microfacet, thus, only those microfacets that are not shadowed by others are assumed to contribute to the local reflectance [36]. In reality, incident light will bounce multiple times within a surfaces microgeometry, meaning even microsurfaces that aren't visible directly from the light will exhibit some reflectance. This is known as *interreflection*. Surfaces can look overly dark and conservation of energy will be violated when this phenomena is not modelled [37]. Figure 1.6 illustrates masking and shadowing. The appropriately named *masking-shadowing function*, $G(\mathbf{l}, \mathbf{v}, \mathbf{m})$, accounts for the effects of masking and shadowing. It provides the fraction of microfacets with normal $\mathbf{m}$ that are visible from directions $\mathbf{v}$ and $\mathbf{l}$ [16].

The micro-BRDF, $f_\mu(\mathbf{l}, \mathbf{v}, \mathbf{m})$, describes how light is reflected of an individual microfacet. There are two common choices for micro-BRDF. It's typical in a specular (macro) BRDF, that all the microfacets are modelled as perfect mirrors [38]. This means that the micro-BRDF will reflect an incident light ray $\mathbf{l}$ in only one direction: $\mathbf{l}$ reflected about the microfacet normal $\mathbf{m}$. Conversely, some diffuse (macro) BRDFs model all the microfacets as perfect diffuse surfaces [39]. As explained in section 1.2.1, in compliance with Lambertian reflection, these surfaces diffuse incoming light equally in all directions and have no specular component; this translates to a



(a) Some microfacets are masked by others (b) Some microfacets are shadowed by others

Figure 1.6

micro-BRDF of constant value.

The microfacet theory developed in the above paragraphs can be used to create an expression for the overall (macro) BRDF [15]:

$$f(\mathbf{l},\mathbf{v}) = \int_{\mathbf{m}\in\Omega} f_\mu(\mathbf{l},\mathbf{v},\mathbf{m})G(\mathbf{l},\mathbf{v},\mathbf{m})D(\mathbf{m})\frac{(\mathbf{m}\cdot\mathbf{l})^+}{|\mathbf{n}\cdot\mathbf{l}|}\frac{(\mathbf{m}\cdot\mathbf{v})^+}{|\mathbf{n}\cdot\mathbf{v}|}\,d\mathbf{m} \tag{1.9}$$

Again, Ω is the unit hemisphere centred on $\mathbf{p}$ and orientated along the surface normal $\mathbf{n}$. Heitz gives a thorough explanation as to how this equation is derived [36]. Although not used directly in rendering, given a specific choice of micro-BDRF, equation 1.9 can be simplified to obtain a BRDF suitable for use in a shading model [15]. An example of such a simplification can be seen in section 1.2.5.

1.2.5 Specular BRDFs

Physically based specular BRDFs are built upon microfacet theory, and therefore utilise equation 1.9. This equation integrates over all possible microfacet normals $\mathbf{m}$ that point above the macrosurface. However, because each microfacet is modelled as a perfect mirror, $f_\mu(\mathbf{l},\mathbf{v},\mathbf{m})$ will only be non-zero when $\mathbf{m}$ is of a value such that $\mathbf{l}$ is reflected exactly in direction $\mathbf{v}$. The only instance of $\mathbf{m}$ that satisfies this is when $\mathbf{m} = \mathbf{h}$, the half vector introduced in section 1.2.1. Therefore, we can remove the integration in equation 1.9, simplifying it to the case when $\mathbf{m} = \mathbf{h}$. After some further derivation we arrive at the following equation for specular BRDFs [36]:

$$f_{spec}(\mathbf{l},\mathbf{v}) = \frac{F(\mathbf{h},\mathbf{l})G(\mathbf{l},\mathbf{v},\mathbf{h})D(\mathbf{h})}{4|\mathbf{n}\cdot\mathbf{v}||\mathbf{n}\cdot\mathbf{l}|} \tag{1.10}$$

All the terms in equation 1.10 are familiar, although the Fresnel reflectance, $F(\mathbf{h},\mathbf{l})$, is parametrised slightly different to the one introduced in section 1.2.4 - this is explained in the section below. Given this equation, the construction of a specific specular BRDF then boils down to choosing how these functions are implemented.

Fresnel Reflectance Implementations

As explained in section 1.2.4, we don't make use of the Fresnel equations directly, but rather utilise approximations. The following functions are given in terms of $\mathbf{n}$ to maintain consistency with much of the literature, but note that in practice $\mathbf{n}$ is replaced with $\mathbf{h}$ as stipulated by equation 1.10. An early approximation was given by Cook and Torrance [38]. Schlick then improved upon

this with his own function that has since seen widespread use [34]:

$$F(\mathbf{n},\mathbf{l}) = F_0 + (1 - F_0)(1 - (\mathbf{n}\cdot\mathbf{l})^+)^5 \tag{1.11}$$

This function approximates the Fresnel reflection with less than a 1% error [34]. Figure 1.7 shows the difference between the actual Fresnel reflectance and this approximation for several materials. The equation interpolates between F_0 and white in a non-linear manner that emulates the description given in section 1.2.4. When the light is directly incident to the surface, $\mathbf{n}\cdot\mathbf{l} = 0$ so the Fresnel reflectance will be F_0. As the angle between the microfacets' normal and the incident light increases, $\mathbf{n}\cdot\mathbf{l}$ decreases and so the overall value of F tends towards white.

Lagarde created a more optimised version of Schlick's approximation by using *Spherical Gaussians* [40]:

$$F(\mathbf{n},\mathbf{l}) = F_0 + (1 - F_0)2^{(-5.55473(\mathbf{n}\cdot\mathbf{l})-6.98316)(\mathbf{n}\cdot\mathbf{l})} \tag{1.12}$$

Unreal engine makes use of this formulation for their Fresnel term [9]. Frostbite on the other hand, have adapted the Schlick approximation into a more flexible form [10]:

$$F(\mathbf{n},\mathbf{l}) = F_0 + (F_{90} - F_0)(1 - (\mathbf{n}\cdot\mathbf{l})^+)^{\frac{1}{p}} \tag{1.13}$$

F_{90} can be used to define the colour that the reflectance tends towards at glancing angles - it doesn't have to be white. Although this is physically incorrect, it does give artists more control. The p variable is used to modify the steepness of the transition to F_{90}.

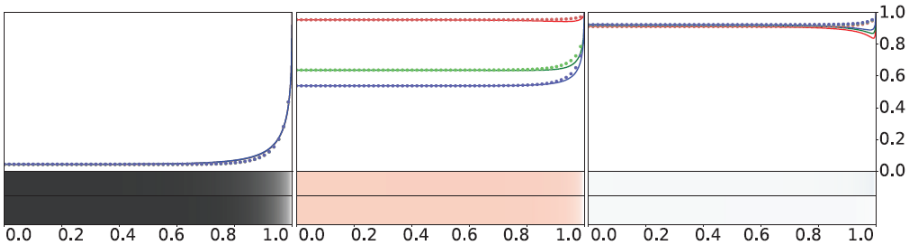


Figure 1.7: From left to right the materials are, glass, copper and aluminium. Along the x-axis is plotted $\sin(\theta_i)$, and along the y-axis is the intensities of each RGB channel. The solid lines are the actual Fresnel reflectance, and the dotted lines are the values given by the Schlick approximation. Similarly, the upper colour strip is the actual Fresnel reflectance, and the lower is that yielded from the Schlick approximation. Taken from [15].

NDF Implementations

In this section we will only concern ourselves with *isotropic* NDFs, which are those that are invariant when rotated about n. *Anisotropic* NDFs do exist and are crucial when rendering surfaces like brushed metal. A prominent isotropic NDF is the Beckmann distribution [41]. It is widely used in the field of optics and is the NDF of choice in the Cook-Torrance BRDF [42] [38]. Walter et al give an equation for the Beckmann distribution [42]. After using some identities to manipulate it, the trigonometric functions can be replaced with dot products to yield the following:

$$D(\mathbf{m}) = \frac{\mathcal{X}^+(\mathbf{m}\cdot\mathbf{n})}{\pi\alpha_b^2(\mathbf{m}\cdot\mathbf{n})^4}\exp\left(\frac{(\mathbf{m}\cdot\mathbf{n})^2-1}{\alpha_b^2(\mathbf{m}\cdot\mathbf{n})^2}\right) \tag{1.14}$$

α_b controls the roughness of the surface and thus the spread of the specular response [38].

The other NDF that has seen extensive use in the graphics community is the GGX distribution. Originally formulated by Trowbridge and Reitz in 1975, the GGX distribution only started to see use after it was reformulated by Walter et al in 2007 [43][42]. It has the following form:

$$D(\mathbf{m}) = \frac{\alpha_g^2\mathcal{X}^+(\mathbf{m}\cdot\mathbf{n})}{\pi(1+(\mathbf{m}\cdot\mathbf{n})^2(\alpha_g^2-1))^2} \tag{1.15}$$

α_g is similarly used to control the roughness of the surface, although it scales in a different manner. Burley proposes mapping $\alpha_g = r^2$ where r provides a perceptually linear change in roughness as it moves between 0 and 1 [12]. Exposing parameters to artists in this linear manner is a common trope in graphics. Karis outlines it as one of the key goals they pursued when migrating the Unreal Engine over to a PBR pipeline [9].

Masking-shadowing Function Implementations

Implementations of the masking-shadowing function depend on the NDF used. There are two main options to choose from. The first is a function created by Torrance and Sparrow, which considers adjacent microfacets to form a "V-groove cavity" [44]. The second option is the Smith masking-shadowing function [45]. Heitz thoroughly analyses the two models and concludes that whilst they are both physically based, the Smith function is more accurate to real materials, with the Torrence-Sparrow function producing too small a specular component for rough materials viewed at glancing angles [36]. The most accurate version of the Smith function is the *Smith height-correlated masking-shadowing function*. This takes into account the height of surface points relative to the rest of the surface. This is significant because points that are lower in the surface are more likely to experience masking and shadowing [15].

The Smith height-correlated masking-shadowing function that is intended for use with the GGX NDF is [15]:

$$G(\mathbf{l},\mathbf{v},\mathbf{m}) = \frac{\mathcal{X}^+(\mathbf{m}\cdot\mathbf{v})\mathcal{X}^+(\mathbf{m}\cdot\mathbf{l})}{1+\Lambda(a_\mathbf{v})+\Lambda(a_\mathbf{l})} \tag{1.16}$$

where

$$\Lambda(a) = \frac{-1+\sqrt{1+\frac{1}{a^2}}}{2} \tag{1.17}$$

and

$$a_\mathbf{v} = \frac{\mathbf{n}\cdot\mathbf{v}}{\alpha_g\sqrt{1-(\mathbf{n}\cdot\mathbf{v})^2}} \quad a_\mathbf{l} = \frac{\mathbf{n}\cdot\mathbf{l}}{\alpha_g\sqrt{1-(\mathbf{n}\cdot\mathbf{l})^2}}$$

1.2.6 Diffuse BRDFs

We now focus on diffuse BRDFs, which model the local subsurface scattering component of local reflectance. Similar to the intrinsic specular colour of a material, F_0, we also define an intrinsic diffuse colour called the *subsurface albedo*. Denoted by ρ_{ss}, this is the proportion of transmitted light that is re-emitted at the surface via subsurface scattering. Since this amount is a distribution of wavelengths, ρ_{ss} is represented as an RGB triplet. The more ρ_{ss} tends towards $(0, 0, 0)$, the more absorption takes place within the interior of the associated material.

Diffuse BRDFs can be placed into two categories: those that are built upon microfacet theory and account for surface roughness; and those that don't utilise microfacet theory and just assume the surface is flat. As explained by Akenine-Möller et al, the type of model to use is determined by the differences between the size of the microgeometry, and the subsurface scattering distances [15]. If the microgeometry is larger than the scattering distances, then a rough-surface diffuse BRDF should be used. If the microgeometry is smaller than the scattering distances, then a flat-surface diffuse BRDF can be used.

Flat-surface Diffuse BRDFs

The most common diffuse BRDF is based on Lambertian reflection and has the following formulation:

$$f_{diff}(\mathbf{l}, \mathbf{v}) = (1 - F(\mathbf{h}, \mathbf{l}))\frac{\rho_{ss}}{\pi} \tag{1.18}$$

The π is obtained by setting the *directional-hemispherical reflectance function* to a constant value and integrating it over all values of $\mathbf{v}$ [15]. The inclusion of the Fresnel reflectance ensures that only light that was transmitted into the interior of the object is available to the diffuse term [46]. This helps to comply with the law of conservation of energy. When the $(1 - F(\mathbf{h}, \mathbf{l}))$ is removed from equation 1.18, what is left is commonly referred to as the Lambertian BRDF.

Equation 1.18 has a constant value for all view directions. The physical basis for such a formulation is rooted in the observation that, prior to re-emission, subsurface scattered light will undergo many scattering events in the interior of the object (as explained in section 1.2.2). This has the effect of randomising the directions of outgoing light, which can be intuitively modelled as a constant value over all $\mathbf{v}$ [47]. However, real materials will show some directional bias, whether that's due to refraction or the need to respect Helmholtz reciprocity (which equation 1.18 violates) [15]. Shirley et al present a more accurate diffuse BRDF that is partially coupled with the specular component [24].

Rough-surface Diffuse BRDFs

The most notable microfacet-based, rough-surface diffuse BRDF is that created by Oren and Nayar [39]. It is comprised of: a gaussian NDF; a Torrence-Sparrow masking-shadowing function; and a constant micro-BRDF that models each microfacet as a perfectly diffuse surface. The Oren-Nayar BRDF increases in intensity as $\mathbf{v}$ approaches $\mathbf{l}$, a property that is coherent with the reflectance characteristics of rough surfaces like clay [39]. As a result, their BRDF produces much more realistic results for these rough surfaces when compared to the Lambertian BRDF. Alternative diffuse BRDFs that are based on microfacet theory are given by Gotanda, and another

by Hammon [48][47]. These both make use of the more modern GGX distribution and Smith height-correlated masking-shadowing function.

Burley presents a BRDF that accounts for roughness, but does not use microfacet theory [12]. Instead, it is empirically constructed by observing how light responds to different materials in the *MERL* database. The Frostbite Engine utilises a slightly modified version of Burley's BRDF, where it has been tweaked to respect energy conservation [10].

1.2.7 Illumination

Having thoroughly examined the BRDF, we now study the other function in the reflectance equation, the incoming radiance, $L_i(\mathbf{p}, \mathbf{l})$.

In section 1.2.1, a distinction was made between direct and indirect illumination. Direct illumination is light that arrives directly from a light source. Indirect illumination is light that only strikes point $\mathbf{p}$ after first undergoing collisions with other objects and media within the scene. When considering the unit hemisphere of incoming light, direct illumination contributes high levels of radiance, over small solid angles. Indirect illumination then spans the rest of the hemisphere, with small to moderate levels of radiance. This difference means that when integrating the reflectance equation over all possible incident light directions, it is common to deal with direct and indirect light sources separately.

Indirect Illumination

Indirect illumination impinging upon $\mathbf{p}$ can be computed using *Global Illumination* (GI) or *Local Illumination* techniques. GI algorithms determine $L_i(\mathbf{p}, \mathbf{l})$ by backtracking, and explicitly simulating the previous light-matter interactions that have led to that illumination. Such techniques are dependent on having a global view of the objects in the scene at each shading point. The ray tracing algorithm briefly discussed in the introduction is a GI technique. The tremendous amount of realism and detail that GI techniques yield, unfortunately demand an equally tremendous amount of compute time. The frame times given for ray tracing was indicative of this. As a result, GI algorithms cannot be computed per frame in real-time rendering. However, for static environments, GI can be used to precompute some aspects of lighting, which are then retrieved at run time. This process is known as *baking* [49].

Local Illumination techniques do not require a global view of scene objects; they only need to be aware of the current object that is being shaded. We have already examined a crude Local Illumination technique for modelling indirect illumination: the ambient term presented in section 1.2.1. More advanced techniques are Image Based Lighting and Irradiance Mapping [21][50].

Direct illumination

Computing the direct illumination incident to $\mathbf{p}$ involves integrating the reflectance equation over specific, known values of $\mathbf{l}$. There are three types of light sources: area, punctual and directional. Area lights, as the name suggests, have a size, as well as a location. When projected onto the unit hemisphere, this size translates to a solid angle. Therefore, to work out the contribution of incoming radiance from that area light, an integration is done over values of $\mathbf{l}$ that lie within its projected solid angle.

Punctual lights are similar to area lights, except they have an infinitesimally small size. This simplifies the evaluation of the reflectance equation because only one direction $\mathbf{l}$ needs to be integrated for each punctual light source. However, punctual lights are an abstraction of the real world - all lights occupy some area - and therefore the simplified computation they provide comes at a cost. Point lights will produce very small specular highlights on shiny materials, which looks unrealistic. To combat this, artists will often increase the roughness of the material to spread the highlight out, but in doing so they couple the specific lighting environment and material properties together [10] [9]. In the introduction we discussed why such a practise can be problematic.

Directional lights are then a further simplification, where they are assumed to be positioned so far away, that their direction is constant over all objects in the scene. Thus, they are not associated with a particular location.

Point Lights

The most common form of punctual light is the point light. A point light emits constant radiance over all directions. The colour of the point light is denoted as $\mathbf{c}_{PL}$, which is an unbounded RGB triplet [21]. $\mathbf{c}_{PL}$ is then defined as the reflected radiance from a perfectly diffuse white surface that directly faces the light. Given this definition, Hoffman derives a simplified version of the reflectance equation for a single point light [21]. Extending this for multiple point lights, the reflectance equation becomes:

$$L_o(\mathbf{p}, \mathbf{v}) = \pi \sum_{i=0}^{N-1} f(\mathbf{l}_{PL_i}, \mathbf{v}) \mathbf{c}_{PL_i} (\mathbf{n} \cdot \mathbf{l}_{PL_i})^+ \tag{1.19}$$

where N is the number of point lights.

Light intensity is attenuated as the distance between the light source and shaded point $\mathbf{p}$ increases. For point lights, this attenuation is equal to the inverse of the squared distance. Therefore, $\mathbf{c}_{PL_i}$ will be proportional to $1/distance^2$. The dot product term has been clamped to 0, so that the contribution of point lights that are located below the surface is discarded.

1.2.8 Dynamic Range and Displays

When rendering, a distinction can be made between the *dynamic range* of values produced via shading, and the dynamic range supported by the display. Dynamic range is defined as the ratio between the highest and lowest brightness values. Most displays are only capable of *Low Dynamic Range* (LDR), which means the maximum value of RGB triplets that they can display is restricted to 1 for each channel. However, real world brightness values, which shading seeks to emulate, often span a much greater range [51]. This range is called *High Dynamic Range* (HDR). RGB triplets that measure HDR values have practically no restriction on the upper values for their components. It is therefore necessary to provide a mechanism that converts between HDR RGB triplets and LDR RGB triplets. This conversion process is called *tone mapping* [52]. A good *tone mapping operator* seeks to recreate on the display, the perceptual impression that a viewer would experience if they were present in the real life scene [52]. This is known as *image reproduction*.

Tone Mapping

In a non-physically based renderer, where the Blinn-Phong shading model might be employed, the discrepancy is dealt with by simply performing all shading calculations in LDR, and passing the values directly to the display. This behaviour is essentially equivalent to employing a tone mapping operator that clamps all RGB values to 1 per channel. As explained above, the dynamic range of the real world far outstrips LDR, so this necessarily hampers the shading ability of non-physically based renderers. Either they artificially compress the brightness range in the scene, or all the bright pixel values just become white, with the detail between them being lost. Neither approach is desirable.

A physically based renderer is focused on simulating real world light-matter interaction, so Physically Based Shading is done in the HDR space of RGB triplets. This is why the $\mathbf{c}_{PL}$ RGB triplet presented in section 1.2.7 was unbounded. This ensures that the vast brightness differences that exist between elements in a scene - for example, the vast difference that will exist between the radiance of the sun and that of a weak point light - can be accurately represented when shading. Once shaded, the HDR RGB triplet is tone mapped to obtain the LDR value that is submitted to the display. Crucially, this tone mapping is done in a more advanced way than the clipping above, with image reproduction being the goal. All tone mapping operators that have this aim are based around the *sigmoid function* [52]. This function provides a smooth roll off of values, so that some of the detail in the darkest and brightest elements of the frame is preserved.

One of the early tone mapping operators used in real-time rendering is given by Reinhard et al [51]. More recently, the use of *filmic* operators, that emulate the image reproduction properties of photographic film, has been widely adopted. This was spearheaded by Hable, who created an operator that was first used in the game Uncharted 2 [53]. Building on this concept, the *Academy Colour Encoding System* (ACES) is a standard that prescribes how colour should be managed in movies and TV. It has since spread into real-time rendering, being the tone mapping operator of choice in the Unreal Engine [54].

Exposure

A closely related topic to tone mapping is *exposure*. Exposure is used to scale HDR RGB triplets before they are tone mapped; it has the effect of altering the overall brightness level of a rendered frame. Exposure can be set statically per scene, or computed dynamically by statistically analysing the brightness levels of previous frames [15].

Gamma Correction

Humans do not perceive radiance in a linear manner, with absolute differences in lower values being much more noticeable than the same differences in higher values. This means that if a human is given a uniform range of radiance values, they will perceive it as entirely non-uniform in its brightness - colours will be seen to rise very quickly towards white. Displays compensate for this perceptual phenomena by mapping input radiance values to non-linear output values. For most displays, this mapping is done in accordance with the *sRGB display transfer function*. In this way, a display outputs colours that are perceived as uniform by humans. However, when rendering, we want the display to output the exact radiance values we give to it, since these have

been accurately calculated via shading. Therefore, as the final step before colours are submitted to the display (after tone mapping and exposure have been applied), we apply the inverse of the sRGB display transfer function to all radiance values. This is known as *gamma correction*. Gritz and d'Eon detail the artifacts that can occur if gamma correction is not performed [55]. Gamma correction can be approximated by raising all radiance values to the power of 2.2, or done more accurately by using the exact inverse of the sRGB display transfer function [10].

Chapter 2

Methods

This chapter documents the design and implementation of the renderer. Details of the comparison between Blinn-Phong shading and PBS is given in the following chapter.

2.1 Design and Implementation

This section provides a general overview of the design and implementation of the renderer. We begin by discussing the language, technologies and libraries used to develop the program. This is followed by an explanation of the architecture, which has particular focus on how the two different shading models were supported in the same application. Finally, we detail how the development was carried out, providing information on the development tools used and the software engineering methodology adopted.

2.1.1 Technologies Used

The renderer is written in C++. This choice was made for a number of reasons. First and foremost, C++ is a language I'm familiar and proficient with, having used it extensively for my own personal projects. Furthermore, real-time rendering is by it's very definition, highly performance orientated, which excludes all but the fastest programming languages. Finally, C++ is the most widely used language in the graphics industry, and as such is widely supported by graphics APIs, and a multitude of useful libraries. The C++17 standard is used as it is modern, supported by many compilers and offers helpful features like the `std::filesystem` library and structured bindings.

Like any renderer, the program utilises a graphics API. OpenGL was chosen for its simplicity, and again, due to my familiarity with it. Although Vulkan and Direct3D boast a much richer and more low-level API, this enhanced control was superfluous to requirements so would have been more of a hindrance than a benefit. Version 4.6 of OpenGL is used.

The renderer is supplemented with several libraries, all of which are listed and described in Appendix B.1.

2.1.2 General Architecture

The renderer is split into a few main components. The `Application` class manages the whole program. It is concerned with initialising and shutting down the various subsystems, and also performs the main render loop. The class creates and maintains a `Window` object, registering callback functions with it to capture pertinent events. The `Application` class can be regarded as providing the necessary framework for any type of interactive application to be created. The specifics are then left up to the `Workspace` class. This class acts as a sandbox environment in which resources can be loaded, scenes created and commands given to the renderer. It also holds an instance of the `Camera` class. This camera is designed to emulate the Unity Engine style of editor camera. The `Scene` class is a simple way of assembling together models and lights into a

3D scene. Models can be imported using the `Model` class, or more simplistic meshes, including cubes and spheres, can be created using the `ModelFactory` class.

The rendering API is exposed by the `Renderer` class. This includes methods for common utility commands, like setting the clear colour, and drawing commands, such as those for rendering individual models and others for rendering entire scenes. However, the `Renderer` class doesn't perform any of the actual rendering itself. Instead, depending on the active renderer type, it forwards any rendering calls onto one of the two renderer implementations: either the `BlinnPhongRendererImplementation` is invoked, or the `PBRRendererImplementation`. Both feature the same API because both derive from the `RendererImplementation` pure virtual class, but they behave very differently. As the names would suggest, the `BlinnPhongRendererImplementation` is a renderer that makes use of the Blinn-Phong shading model, whilst the `PBRRendererImplementation` is a physically based renderer. The Blinn-Phong renderer is covered in section 2.2, and the details of the physically based renderer, including the specific physically based shading model that is used, is given in section 2.3. The whole program is designed so that the renderer implementation being used can be swapped during runtime, making it easy to compare the two approaches.

The two renderer implementations, as well as other parts of the program, make extensive use of the constructs provided by OpenGL. To this end, considerable time was spent abstracting away elements of the OpenGL API behind coherent, clear interfaces. These interfaces include provisions for creating and manipulating vertex buffers, index buffers, framebuffers, textures and shaders. Not only do they serve to create cleaner code, but the interfaces also have the added benefit of hiding the notoriously confusing, state machine nature of OpenGL, behind the more standard OOP paradigm.

2.1.3 Development Process

Development was undertaken in an agile manner, being driven by a Kanban board hosted on an Azure DevOps project. A number of days at the start of development were dedicated to assessing the required work, and packaging that into a set of clear, actionable work items. These were arranged hierarchically so that overarching milestones could be identified. Throughout development, new work items were added, old ones were removed when they fell out of scope, and invaluable documentation was accumulated in the items that remained.

Git version control was used and the repo was hosted on the same Azure DevOps project. For each modification I made to the program, I checked out a new branch, and when finished with development, created a pull request. Although I only worked on my own, I found the pull request feature in Azure DevOps to be extremely useful. It allowed me to easily and thoroughly review my changes before I merged them in, which on many occasions meant I identified and solved a bug which otherwise would have made its way into the code base. Had I been simply committing my changes to the main branch, I don't think I would have found it as easy to do that due diligence.

Visual Studio was my IDE of choice. It's an application I have lots of experience with, and one that provides an impressive suite of debugging tools. However, I decided not to directly use the Visual Studio build system, instead opting for Premake. Premake allows for the specification of projects to be done in an intuitive, and build system agnostic way. I found building with Premake to be much easier, and if I choose to migrate the program to other platforms in the

future, it should make that process simpler.

Debugging graphics applications is incredibly difficult since most of the important stuff happens on the GPU. Therefore, I also utilised RenderDoc, which is a tool that allows one to inspect an application as it moves through all the stages of the graphics pipeline.

The testing process for the application was greatly aided by being able to draw on two renderer implementations. When developing the physically based renderer, I often compared its renders to that of the Blinn-Phong renderer to ensure that nothing was fundamentally incorrect[1].

2.2 Blinn-Phong Renderer Implementation

As previously discussed, the Blinn-Phong renderer is responsible for rendering scenes using the Blinn-Phong shading model.

2.2.1 Operation of the Renderer

Upon program initialisation, the Blinn-Phong renderer creates a few notable items. A framebuffer is created which stores the output fragment colours of the shader program. It is configured to perform *multisampling*, which is a simple anti-aliasing technique [56]. The visual enhancements this yields are judged to be well worth the performance costs it incurs. The key feature that distinguishes this framebuffer from the one used in the physically based renderer, is that it can only store LDR RGB triplets. This is realised in practice by specifying the framebuffer's colour attachment format as `GL_RGBA8`. Restricting the renderer in this manner is consistent with the description given in section 1.2.8, on how tone mapping is dealt with in non-physically based renderers.

The actual shader program that implements the Blinn-Phong shading model is also created. The program is specified in the OpenGL Shader Language (GLSL), and the source code is read in from external files. This shader program is explored in detail in section 2.2.2.

Following initialisation, the renderer is ready to draw scenes to the screen. When a scene is submitted, several actions take place. First, all the uniforms (modifiable variables within the shader program) that are constant over the whole scene are set. These include details of all the point lights, as well as camera transforms. Secondly, all the models in the scene are iterated through and a draw call, which invokes the shader program, is submitted for each. During this process, uniforms that specify materials and model transforms will be be constantly changing. Following this, all that is left to do is to output the contents of the framebuffer to the display. This is accomplished by 'blitting' the Blinn-Phong framebuffer to the default framebuffer that is used by the windowing system.

2.2.2 Shader Program

The shader program begins by calculating variables that will be used throughout the shading process. Of particular interest is how the surface normal $\mathbf{n}$ is obtained. In both this shader program and that used in the physically based renderer, the normal is either taken from vertex attributes, or, if specified, from a *normal map*. The normal map is a modern method of *bump*

[1]The more specific aspects of the physically based renderer obviously could not be tested in this way since the two implementations still use very different shading models

mapping, which was first introduced by Blinn in 1978 [57]. The normal map is used to specify $\mathbf{n}$ per pixel, which greatly increases the amount of surface detail that can be rendered.

The Blinn-Phong shading model is then utilised. The ambient lighting component is calculated using equation 1.4 with $x = 0.02$. Then for each point light in the scene, the diffuse and specular terms that are set out in equation 1.7 are evaluated. These values are all summed to get the colour of the fragment. Since the code that implements the Blinn-Phong shading model is so similar to the equations given previously, no further discussion is given here. However, all the shader code used for both the Blinn-Phong and physically based renderers can be viewed in Appendix D.

Prior to that colour being outputted to the framebuffer, gamma correction is performed, which was discussed in section 1.2.8. Lagarde and de Rousiers outline the two common ways in which gamma correction is carried out, and warn against using the approximate approach since it introduces too much error in darker colours [10]. Heading this advice, both the Blinn-Phong and physically based shader programs perform gamma correction using the exact inverse of the sRGB display transfer function. The code for this is given in Listing 1.

```
vec3 gammaCorrectColor(vec3 color)
{
        vec3 SRGBEncodedHigher = (1.055f * pow(color, vec3(1.0f / 2.4f))) - 0.055f;
        vec3 SRGBEncodedLower = 12.92f * color;
        float rSRGBEncoded = (color.r > 0.0031308f) ? SRGBEncodedHigher.r : SRGBEncodedLower.r;
        float gSRGBEncoded = (color.g > 0.0031308f) ? SRGBEncodedHigher.g : SRGBEncodedLower.g;
        float bSRGBEncoded = (color.b > 0.0031308f) ? SRGBEncodedHigher.b : SRGBEncodedLower.b;
        return vec3(rSRGBEncoded, gSRGBEncoded, bSRGBEncoded);
}
```

Listing 1: Shader code for performing gamma correction

2.2.3 Material Specification

As was explained in detail in chapter 1, the appearance of an object depends heavily on the properties of its surface. These properties are encapsulated within a material. The 3D models that make up a scene are formed from multiple meshes, with each mesh being associated with a material. Materials that are supplied to the Blinn-Phong renderer are of the form given in Table 2.1. The global and per pixel variants of the same attributes are not intended to be used in conjunction. For example, either a `diffuseColor` or `diffuseMap` is specified - not both.

2.2.4 Lighting

Scenes submitted to the Blinn-Phong renderer contain point lights. Each point light is defined by the following attributes: `worldPosition`, `diffuseComponent`, `specularComponent` and `lightRadius`. As discussed in section 1.2.7, point lights are a type of punctual light, so have a location, hence `worldPosition`. The `diffuseComponent` attribute maps to the $\mathbf{c}_{light_{diff}}$ variable in equation 1.7, and similarly, the `specularComponent` maps to $\mathbf{c}_{light_{spec}}$. Note that these two

Attribute	Data Type	Description
`diffuseColor`	`glm::vec4`	Specifies the diffuse colour globally over the whole mesh. Maps directly to the $\mathbf{c}_{surface_{diff}}$ variable in equation 1.7. The fourth component in the vector represents transparency.
`diffuseMap`	`Texture`	Specifies the diffuse colour per pixel.
`specularColor`	`glm::vec3`	Specifies the specular colour globally over the whole mesh. Maps directly to the $\mathbf{c}_{surface_{spec}}$ variable in equation 1.7.
`specularMap`	`Texture`	Specifies the specular colour per pixel.
`shininess`	`float`	Specifies the shininess of the mesh. Maps directly to the $surface_{shininess}$ variable in equation 1.7.
`normalMap`	`Texture`	Optional attribute for specifying surface normals per pixel.

Table 2.1: The specification of a Blinn-Phong material

values are LDR RGB triplets so are limited to 1 for each channel, unlike the physically-based point lights described in section 1.2.7. Finally there is the `lightRadius` attribute, which is used when calculating how much a light is attenuated with distance. Both the Blinn-Phong and physically based shader programs use the same attenuation code, and the nature of it is much more physically based than what is typical for use in a non-physically based shader. With this in mind, discussion of how light attenuation is implemented is deferred to section 2.3.4.

2.3 Physically Based Renderer Implementation

The physically based renderer makes use of a physically based shading model. Its behaviour is defined by the specific model that is employed. As explained in section 1.2.3, a physically based shading model computes the reflectance equation with a physically based BRDF and a physically based incoming radiance function. Section 2.3.2 deals with the choice of BRDF, outlining the details of its construction, justifying design decisions made, and explaining how it is implemented in the shader program. The choice of BRDF then informs the material specification, which is covered in section 2.3.3. The specifics of the lighting model, and how it yields physically based values for the incoming radiance, is discussed in section 2.3.4. Section 2.3.5 explains how tone mapping is implemented, and the specific operator that is used. However, before all that, a description of how the renderer operates is given.

2.3.1 Operation of the Renderer

At a high level, the physically based renderer operates similarly to its Blinn-Phong counterpart, with just a few important differences. Upon initialisation, the physically based renderer creates two framebuffers. These framebuffers store HDR RGB triplets, with their colour attachment format being `GL_RGBA16F`. Having two of these framebuffers facilitates the separation of shading from *post-processing*. Post-processing refers to a collection of screen space effects that can be applied to shaded frames - for our purpose, this includes applying exposure, tone mapping and gamma correction. Separating these two stages out is a common pattern used by many renderers.

Two shader programs are created on startup, one that carries out the PBS, and another for doing the post processing steps listed above. When a scene is submitted for rendering, the

physically based renderer makes an additional draw call when compared to the Blinn-Phong renderer. This draw call invokes the post-processing shader program, which after completion, writes the output pixel values to the default framebuffer, displaying the result on the screen.

2.3.2 BRDF Choice and Implementation

Sections 1.2.5 and 1.2.6 enumerated several options for specular and diffuse BRDFs. From these options, a specific BRDF was chosen for the physically based renderer.

Specular BRDF

The specular BRDF is built upon equation 1.10, so the specific choice of Fresnel reflectance, NDF and masking-shadowing function are explained below.

The Fresnel reflectance is calculated using the Schlick approximation, which is given by equation 1.11. It has the benefits of being simple to compute, and accurate. Using the more flexible variant of the Schlick approximation (equation 1.13) would serve to slightly increase the variety of substances that can be represented. However, it would also greatly increase the complexity of the material model, hence why the more basic variant is favoured. Due to my unfamiliarity with the theory behind Spherical Gaussian approximations, Lagarde's optimised version of the Schlick approximation (equation 1.12) is not used. The implementation of the Schlick approximation is given in Listing 2.

```
vec3 calculateFresnelSchlickApproximation(vec3 f0, float u)
{
        return f0 + (1 - f0) * pow((1 - u), 5.0f);
}
```

Listing 2: Shader code for calculating the Fresnel reflectance using the Schlick approximation

In Burley's observations of real world materials, he points out that most surfaces exhibit specular lobes with long tails [12]. Furthermore, he notes that traditional NDFs, such as the Beckmann distribution, do not model these tails well, whereas the GGX NDF is much more accurate. Therefore, knowing it to be more representative of the real world, the GGX distribution is chosen as the Normal Distribution Function. The implementation of the GGX NDF is based of equation 1.15 and can be seen in Listing 3. Two things are noteworthy. The first is that the Burley's mapping of α to a linear roughness value is being employed. The second is that the $\mathcal{X}^+(\mathbf{m} \cdot \mathbf{n})$ term has not been carried over to the implementation. This is because that term will only evaluate to 0 when the half vector $\mathbf{h}$ points below the surface (recall that $\mathbf{m} = \mathbf{h}$ in a microfacet based specular BRDF). This only happens when $\mathbf{l}$ or $\mathbf{v}$ point below the surface. When $\mathbf{l}$ points below the surface, the whole reflectance equation is set to 0 (due to the $(\mathbf{n} \cdot \mathbf{l})^+$ term), and when $\mathbf{v}$ points below the surface, the depth testing stage of the graphics pipeline will discard the shaded fragment [2]. Therefore, the $\mathcal{X}^+(\mathbf{m} \cdot \mathbf{n})$ term is redundant in practice and consequently omitted.

As previously discussed, Heitz proved that the Smith masking-shadowing function is more

[2] normal mapping can cause issues with this assumption but we ignore such cases for simplicity

```
float calculateGGXDistribution(float nDotH)
{
        float alpha = g_materialProperties.roughness * g_materialProperties.roughness;
        float alpha2 = alpha * alpha;
        float x = 1 + (nDotH * nDotH * (alpha2 - 1.0f));
        return alpha2 / (PI * x * x);
}
```

Listing 3: Shader code for the GGX NDF

accurate than the Torrance and Sparrow function. Therefore, the Smith height-correlated masking-shadowing function is used. However, instead of directly implementing the GGX form of the function given in equation 1.16, an optimisation is utilised. Karis presents an approximation for the Smith masking function (a less accurate version of the Smith function that only accounts for masking) [9]. Hammon observed that when using this approximation, the Smith height-correlated function for the GGX distribution has terms that cancel out with the denominator of the general specular BRDF formula (equation 1.10) [47]. He gives the following equation:

$$\frac{G(\mathbf{l},\mathbf{v},\mathbf{h})}{4|\mathbf{n}\cdot\mathbf{v}||\mathbf{n}\cdot\mathbf{l}|} \approx \frac{1}{2 * lerp(2|\mathbf{n}\cdot\mathbf{l}||\mathbf{n}\cdot\mathbf{l}|, |\mathbf{n}\cdot\mathbf{l}| + |\mathbf{n}\cdot\mathbf{l}|, \alpha_g)} \tag{2.1}$$

Hammon compares this approximation to the exact Smith height-correlated masking-shadowing function and finds only minor differences. Due to this and the performance improvements the approximation yields, Hammon's approximation is used to implement the shadowing function. See Listing 4 for the shader code.

Because of the inclusion of the specular BRDF's denominator, using the Hammon approximation also has implications for the overall form of the specular BRDF. The overall specular BRDF is given by Listing 5; note that there is no denominator present.

```
float calculateHammonSmithMaskingSpecularDenominatorAppoximation(float nDotL)
{
        float alpha = g_materialProperties.roughness * g_materialProperties.roughness;
        float x = 2.0f * abs(nDotL) * abs(g_dotProducts.nDotV);
        float y = abs(nDotL) + abs(g_dotProducts.nDotV);
        return 1.0f / (2.0f * mix(x, y, alpha));
}
```

Listing 4: Shader code for the Hammon approximation of the Smith height-correlated masking-shadowing function

```
vec3 specularTerm = fresnelReflectance * hammonSmithMaskingSpecularDenominatorApproximation *
↪  NDF;
```

Listing 5: Shader code for the Specular BRDF

Diffuse BRDF

The diffuse BRDF used is the Lambertian BRDF with the Fresnel reflectance term, which is given by equation 1.18. Although BRDF's that take into account surface roughness, such as the Burley diffuse BRDF, are technically more accurate to the real world, the visual differences with the Lambertian BRDF are often subtle [9]. Couple this with the simplicity of the Lambertian BRDF, and it becomes an attractive choice. The implementation of the diffuse BRDF is given in Listing 6. The only difference between this and equation 1.18, is the additional multiplication by `(1.0f - g_materialProperties.metalness)`. This term is required to account for the fact that metals absorb all transmitted light so have no diffuse component - when `g_materialProperties.metalness` is set to 1, the diffuse component will have no effect. The metalness material property is explained in section 2.3.3.

```
1 vec3 diffuseTermContribution = (vec3(1.0f) - fresnelReflectance) * (1.0f -
  ↪  g_materialProperties.metalness);
2 vec3 diffuseTerm = diffuseTermContribution * (g_materialProperties.baseColor / PI);
```

Listing 6: Shader code for the diffuse BRDF

2.3.3 Material Specification

The shading model detailed above requires the presence of some material properties. These properties are encapsulated within the material specification for the physically based renderer, which is described in Table 2.2. Again, the global and per pixel variants of the same attribute are not used together. This material specification is a subset of that used in the Disney Principled model [12].

The motivation for representing both the ρ_{ss} and F_0 values through the one `baseColor` attribute is that is saves memory. This is particularity the case when the texture variants are used. However, using this representation does necessitate extra processing to retrieve the two values, and also an additional material parameter to be present, `metalness`. `metalness` only makes physical sense when it is either set to 0 or 1 - it is not possible for a material to be partially a metal. Most materials stick to this convention. However, permitting `metalness` to roam in between these limits can help to smooth transitions from metals to dielectrics. This could be utilised when, for example, representing the boundary between metal and rust.

Section 2.3.2 and Listing 6 already outlined how ρ_{ss} is retrieved from the `baseColor` using `metalness`. Listing 7 shows how the retrieval of F_0 is implemented. As discussed in section 1.2.4, $F_0 = (0.04, 0.04, 0.04)$ is a suitable assignment for dielectrics.

Further optimisations could be made to this material specification. A popular option is to pack the `roughnessMap` and `metalnessMap` into the same texture, but at separate channels. The *glTF* 3D model format makes use of this [58]. However, importing 3D models of a different format is then made more difficult, so packing is not used in the material specification.

Attribute	Data Type	Description
`baseColor`	`glm::vec4`	For dielectrics it specifies the subsurface albedo ρ_{ss}, and for metals it specifies the intrinsic specular colour F_0. The fourth component in the vector represents transparency. This value is specified globally over the whole mesh.
`baseColorMap`	`Texture`	Specifies the base colour per pixel.
`roughness`	`float`	Specifies the surface roughness globally over the whole mesh. Following Burley's linear remapping suggestion, this value is between 0 and 1.
`roughnessMap`	`Texture`	Specifies the surface roughness per pixel.
`metalness`	`float`	Specifies the metalness globally over the whole mesh. Takes on values between 0 and 1, where 0 means a dielectric, and 1 means a metal.
`metalnessMap`	`Texture`	Specifies the metalness per pixel.
`normalMap`	`Texture`	Optional attribute for specifying surface normals per pixel.

Table 2.2: The specification of a physically based material

```
1 const vec3 F0_FOR_DIELECTRICS = vec3(0.04f);
2 ...
3 g_materialProperties.f0 = mix(F0_FOR_DIELECTRICS, g_materialProperties.baseColor,
  ↪  g_materialProperties.metalness);
```

Listing 7: Extracting F_0 from `baseColor` using `metalness`

2.3.4 Lighting

As discussed in section 1.2.7, evaluating the reflectance equation can be done separately for indirect and direct illumination. Unfortunately, due to time constraints, indirect illumination is only evaluated using a simple ambient term. This term does a basic job of ensuring no shaded areas are completely black, but it is much more empirically based than physically based, and also violates the law of conservation of energy. A much more physically based and accurate solution would have been to use Image Based Lighting and Irradiance Mapping.

Direct illumination is evaluated with the restriction that all light sources are point lights. Although supporting area light sources would be more physically accurate, and avoid the material-light coupling that was detailed in section 1.2.7, dispensing with them allows for the implementation to be greatly simplified.

Computing the reflectance equation is implemented by summing over all the contributions of the point lights in the scene, and then adding on the ambient term. The code for doing this mostly follows equation 1.19. The main difference is that the multiplication by π is not done. This is because π is implicitly embedded in the `luminousPower` attribute of the point light, which is a common practice [59]. The code that calculates the contribution of a single point light is given by Listing 8. `lightRadiance` is the product of two things: an *attenuation factor*, and the point light's radiance when unaffected by attenuation.

```
vec3 BRDFValue = diffuseTerm + specularTerm;
return BRDFValue * lightRadiance * max(nDotL, 0.0f);
```

Listing 8: Calculating the reflectance equation for a single point light

Point Light Specification

A point light that is submitted to the physically based renderer is specified with the following attributes: `worldPosition`, `lightRadius`, `lightColor` and `luminousPower`. The $\mathbf{c}_{PL_i}$ variable in equation 1.19 is calculated in the shader as a function of `lightColor` and `luminousPower`. `lightColor` is an RGB triplet that limited to 1 for each channel, and defines the hue of the point light. `luminousPower` is a float that represents the power of the point light. Together, they allow the radiance value of the point light to take on values in the HDR space.

The use of `lightColor` and `luminousPower` is based of the point light model used by the Frostbite Engine. Lagarde and De Rousiers show that by defining a point light in terms of its luminous power, it is easier to model real-world lights, as this measurement is often given by the manufacturer [10]. From luminous power, an equation for luminous intensity can be derived by integrating over the unit sphere: $luminousIntensity = luminousPower/4\pi$. We then multiply the light colour by the luminous intensity to obtain the light's radiance when unaffected by attenuation. Although we are suddenly using *photometric* rather than radiometric quantities, Lagarde and De Rousiers specify that the calculations are the same for both. See Listing 9 for the code that calculates `lightRadiance`. The way in which `lightAttenuationFactor` is determined is explained below.

```
float luminousIntensity = pointLight.luminousPower / (4.0f * PI);
vec3 lightRadiance = pointLight.lightColor * luminousIntensity * lightAttenuationFactor;
```

Listing 9: Calculating incoming radiance for a point light

Attenuation Factor

As stipulated previously, light intensity attenuates over distance, and for point lights, this attenuation is inversely proportional to the square of the distance between light source and shaded point. However, just calculating the attenuation factor as $1/distance^2$ has three problems. The first is that when the distance is 0, division will cause an error. The second is that as the denominator tends to 0, the attenuation factor will explode to massive values - it should only take on values between 0 and 1. The third is that the attenuation factor will approach 0 asymptotically, never actually reaching it. It is desirable for a light to be fully attenuated at some finite distance, as such lights can be culled, improving performance.

The Unreal and Frostbite Engines give solutions that solve all three of these problems [9][10]. Both of them accomplish this using a similar approach: they offset the denominator, clamp return values, and make use of a windowing function. Frostbite's equation for the attenuation factor is employed in the shader program. It is used over Unreal's because it features a more physical

interpretation for the denominator offset. Listing 10 shows how it is implemented. `lightRadius` is a float that is specified per point light, and gives the maximum distance of a light's reach.

```
float calculatePointLightAttenuationFactor(float lightDistance, float lightRadius)
{
        const float lightSize = 0.01f;
        const uint n = 4;

        float inverseSquaredDistance = 1.0f / pow(max(lightDistance, lightSize), 2.0f);

        float lightDistanceNOverLightRadiusN = 1.0f - pow(lightDistance / lightRadius, n);
        float windowingFunctionValue = pow(clamp(lightDistanceNOverLightRadiusN, 0.0f, 1.0f),
        ↪  2.0f);

        return min(inverseSquaredDistance * windowingFunctionValue, 1.0f);
}
```

Listing 10: Calculating attenuation factor for a point light

2.3.5 Tone Mapping

As previously mentioned, tone mapping is applied in the post processing shader program. As shown by Cerda-Company et al, the Reinhard operator is very effective at image reproduction [60]. However, ACES still excels at image reproduction and we find its filmic response to be more aesthetically pleasing. Therefore, the ACES tone mapping operator is used, and its implementation can be seen in Listing 11. This code utilises some external material that is discussed in Appendix B.2.

Just before tone mapping, exposure is used to scale all the radiance values. This exposure is set as a static value over the scene. Dynamically calculating the exposure is unnecessary here; that is only required in interactive games, where the overall light level can change drastically as the player moves around a scene.

As explained before, the final step of the post processing shader program is to apply gamma correction using the same code that was given in Listing 1.

```
vec3 applyToneMapping(vec3 color)
{
        mat3 inputMatrix = mat3
        (
                0.59719f,  0.07600f,  0.02840f,
                0.35458f,  0.90834f,  0.13383f,
                0.04823f,  0.01566f,  0.83777f
        );

        mat3 outputMatrix = mat3
        (
                1.60475f, -0.10208f, -0.00327f,
                -0.53108f,  1.10813f, -0.07276f,
                -0.07367f, -0.00605f,  1.07602f
        );

        color = inputMatrix * color;
        vec3 a = color * (color + 0.0245786f) - 0.000090537f;
        vec3 b = color * (0.983729f * color + 0.4329510f) + 0.238081f;
        color = a / b;

        return clamp(outputMatrix * color, 0.0f, 1.0f);
}
```

Listing 11: The ACES tone mapping operator

Chapter 3

Results

3.1 Evaluation Strategy

As outlined in the introduction, the focus of this report is on showing how PBS produces more photorealistic images than Blinn-Phong shading. This is fulfilled by highlighting physical phenomena that are modelled more accurately in frames rendered using PBS, than in frames rendered using Blinn-Phong shading. Two such physical phenomena are identified below, and subsequent comparisons performed for each using the application developed.

The first phenomena is the Fresnel effect. Mentioned in section 1.2.4, the Fresnel effect is the observation that a surface becomes more specular at glancing incident light angles. Faul shows that the Fresnel effect is a key contributor to the appearance of specular materials [33]. This suggests that the realism of a shading model is influenced by the degree to which it models the impact of the Fresnel effect. Therefore, it forms a suitable candidate for comparison. In an effort to further build on Faul's investigation, comparing the presence of the Fresnel effect is done by considering the impact it has on the appearance of diffuse materials, rather than specular.

The second physical phenomena is the effect that multiple incident lights have on the appearance of an object. In the real world, the more lights that are incident to an object, the brighter the object will appear. A comparison is performed between the two shading models to ascertain how accurately they depict how a varying number of point lights influence the appearance of an object.

The evaluation is also supplemented with an analysis of FPS. This assesses whether the implemented shading models are indeed sufficiently performant to be used in real time rendering.

3.2 Fresnel Effect Comparison

3.2.1 Testing Method

Testing a shading model's ability to capture the Fresnel effect could be done using a single point light that is incident to an xz-aligned plane. From a constant viewpoint, the point light could then be moved over the plane so as to vary the incident light angle. From that viewpoint, the amount of specular reflection could be observed.

However, the test given here is instead performed by keeping the position of the point light constant, and varying the viewing angle (which is also given in respect to the surface normal). The effect of varying the viewing angle is to vary $\mathbf{v}$, and thus also vary $\mathbf{h}$; varying $\mathbf{h}$ is equivalent to varying the normals of the microfacets that contribute to the specular reflection; varing the normals has the effect of varying the incident light angle on those microfacets. Although this sequence of implications is certainly complex, carrying out the test in this manner is simpler than the alternative approach given above. Shirley et al present a case study that captures the Fresnel effect by also varying the viewing angle [24]. Moreover, as Figure 3.1 shows, this testing approach is justified by conducting observations of the real world: the Fresnel effect causes surfaces to

appear more specular at glancing viewing angles.

For the tests, two scenes have been created. One for the Blinn-Phong renderer, and another for the physically based renderer. Each scene is arranged in the same manner. A diffuse wooden floor is situated at the origin, and a point light is placed just above it. Four viewing angles are considered, 0°, 40°, 65° and 80°. For each of these angles, the outputted frames of the two renderers are sampled. A shading model that captures the Fresnel effect is expected to render the floor with greater specularity as the viewing angle increases (becomes more glancing).

Figure 3.1: The Fresnel effect can be observed in the real world by varying the viewing angle. Taken from [61].

3.2.2 Results

The sampled frames for the two shading models are given in Figure 3.2.

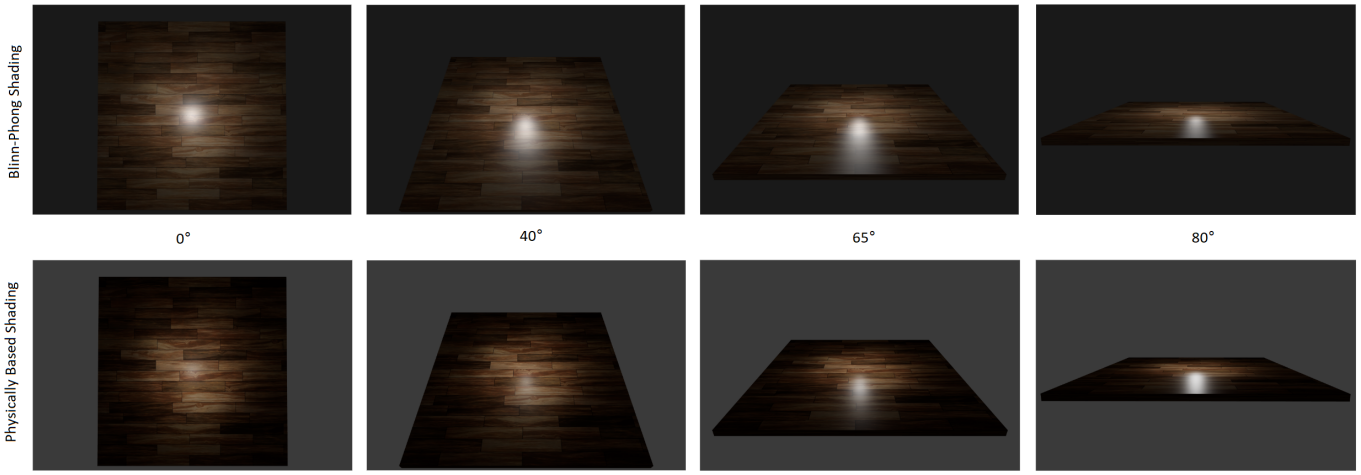


Figure 3.2: Comparing Blinn-Phong and Physically Based Shading for the presence of the Fresnel effect. Viewing angles are given between the images.

3.2.3 Analysis

In the frames rendered using the Blinn-Phong shading model, the wooden floor has a constant specular response over all viewing angles - the Fresnel effect is not being modelled. This is expected, as the Blinn-Phong shading model laid out in equation 1.7 contains no means of computing the Fresnel reflectance. This poses an issue for artists. They could choose material properties that ensure the specular response of an object is accurate at large viewing angles, but this would have the downside of making the object appear much too specular at smaller viewing

angles. This was the approach used for the Blinn-Phong wooden floor material in Figure 3.2. Alternatively, they could do the opposite, and have more accurate specularity at smaller viewing angles, but sacrifice it at large viewing angles. Either way, a compromise is made that limits the realism of frames rendered using the Blinn-Phong shading model.

In contrast, the physically based shading model is clearly exhibiting the Fresnel effect. At small viewing angles, the wooden floor appears mostly diffuse with only a small specular response. As the angle increases towards 90°, so does the specularity increase. Not only this, but the increase also follows the non-linear behaviour described in section 1.2.4. This is evidenced by the fact that the specularity does not change between viewing angles 0° and 40°. It is at 65°, and especially 80°, where an increased specular response is observed. The way in which the specular response varies over the PBS frames bears close resemblance to Figure 3.1. The presence of the Fresnel effect is due to the Fresnel reflectance function in equation 1.10, which is implemented in the physically based shading model by means of the Schlick approximation.

The Fresnel effect is a physical phenomena that is accurately modelled in frames rendered by the physically based shading model, and is completely absent in frames rendered by the Blinn-Phong shading model.

3.3 Multiple Lights Comparison

3.3.1 Testing Method

Similar to the previous test, two scenes have been created for the two renderers, and both are arranged identically. A diffuse wooden wall is put just in front of the camera, and attached to it is a light fixture. A varying number of point lights are then all placed where the bulb of the light fixture would be if it existed.

Each Blinn-Phong point light is given the same properties: the `diffuseComponent` and `specularComponent` attributes are both set to $(1, 1, 1)$; the `lightRadius` attribute is set to 10. Likewise, every PBS point light is given the same properties: the `lightColor` attribute is set to a value of $(1, 1, 1)$; the `luminousPower` attribute is set to 50; the `lightRadius` attribute is set to 10. By setting `luminousPower` to 50, a single PBS point light exhibits an intensity that is similar to a single Blinn-Phong point light.

The test is conducted by varying the number of point lights placed in the scene - from 1 to 10 - and then for each number, sampling the outputted frames of the two renderers.

3.3.2 Results

A subset of the sampled frames for the two shading models are given in Figure 3.3. The pixel values of all the rendered frames were converted to HSV triplets and further analysis was performed. Table 3.1 gives the number of pixels in the rendered frames that were fully saturated, meaning they had a Value component of 255 and were completely white.

3.3.3 Analysis

A number of issues are evident in the frames rendered using the Blinn-Phong shading model. Table 3.1 shows that the number of fully saturated, completely white pixels is substantial. Pixels

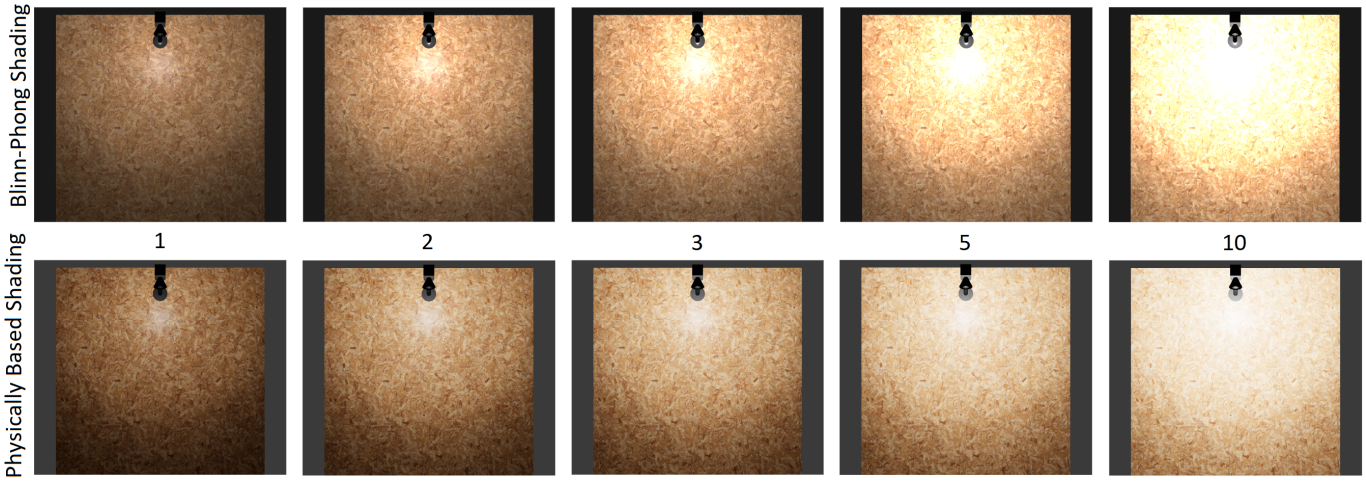


Figure 3.3: Comparing how the Blinn-Phong and physically based shading models render multiple point lights. The number of point lights present in the scene are given between the images.

Point Light Count	Blinn-Phong Shading	Physically Based Shading
1	0	0
2	21028	0
3	83647	0
4	160152	0
5	214592	0
6	249421	0
7	272263	0
8	288192	0
9	301453	0
10	313752	0

Table 3.1: The number of fully saturated pixels in the rendered frames of the two shading models, broken down by number of point lights.

such as these pose a couple of issues. The first is that they are at maximum brightness, so no matter how many additional lights become incident to the surface, the displayed brightness will not increase. Consider the 21028 pixels that are immediately saturated after just two lights are present in the scene. This means a significant portion of the frame becomes no brighter, even after the number of lights is quadrupled. The second issue is that when neighbouring pixels are completely white, all the detail that existed between them is lost. The top row of Figure 3.3 shows that this loss of detail manifests itself as unnatural white void that spreads over the image. All of this behaviour is a consequence of the Blinn-Phong renderer effectively ignoring any shaded values that are outside the LDR space by clipping them.

In contrast, Table 3.1 shows that the frames rendered using the physically based shading model contain no fully saturated pixels. This means all the pixels have space to further increase in brightness as more point lights become incident to the object surface. As a result, Figure 3.3 shows a much more natural increase in brightness on the bottom row. Furthermore, a considerable amount of detail is preserved in the brighter regions. These benefits are realised due to the sophisticated tone mapping that is performed by the physically based renderer. This tone mapping is implemented using the ACES operator, which features a sigmoid shape. The sigmoid causes the shaded HDR RGB triplets to be smoothly tapered off at the extremes, meaning fully saturated

values being outputted to the display are rare.

Overall, it is clear that the physically based shading model renders the effects of multiple incident lights much more accurately than the Blinn-Phong shading model. Using a physically based shading model therefore has important implications for artists. It gives them more freedom to place lights wherever they wish, without worrying that unnatural effects will occur.

3.4 Real Time

3.4.1 Testing Method

The real time capabilities of the Blinn-Phong and physically based shading models are assessed by measuring the FPS produced by their respective renderers. Average FPS is gathered for varying scene complexities. Three test scenes are constructed for each renderer. All of them contain 100 point lights, and the only 3D model used is the light fixture introduced in Section 3.3. The scenes only differ by the amount of light fixtures that are present. By increasing the number of light fixtures, the number of triangles in the scene increases, thereby increasing the complexity. Over a 10 second period, the average FPS is calculated for each test scene, and each renderer.

3.4.2 Results

The average FPS for both renderers is given in Figure 3.4[1]. The scene complexity is expressed using triangle count as that is more expressive than giving the number of light fixture models.

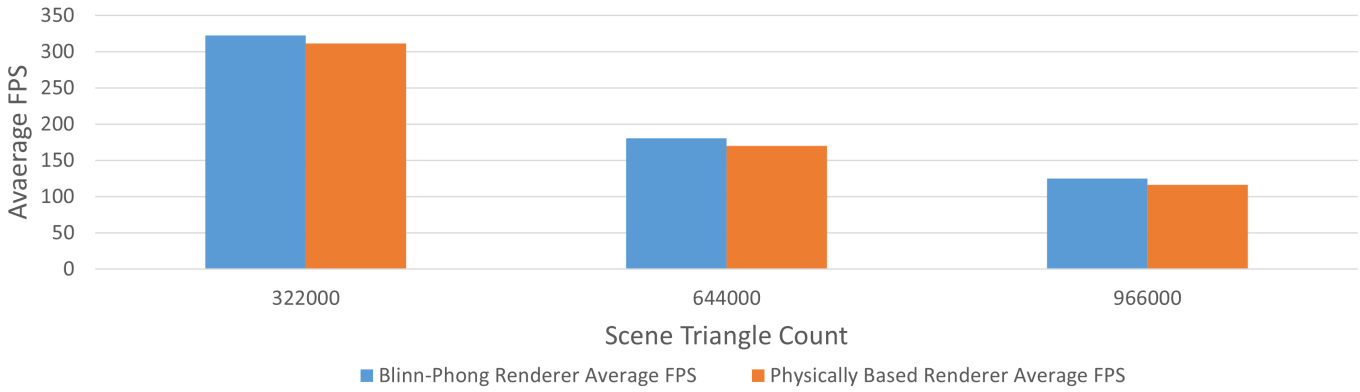


Figure 3.4: Measuring the performance of the Blinn-Phong and physically based renderers for varying scene complexities.

3.4.3 Analysis

In all scenes, the two renderers reported average FPS values that were well above the real time minimum of 30 to 60. This shows that both shading models are suitable for use in real time rendering. Making any further conclusions than this would be unwise, as FPS data is by nature dependent on lots of variables, not just the shading model.

[1]All FPS captures were done with the entire scene in view of the camera. The hardware of the test device consisted of an NVIDIA GeForce RTX 3070 GPU, and an AMD Ryzen 3700X CPU.

Chapter 4

Discussion

<Everything that comes under the 'Results and Discussion' criterion in the mark scheme that has not been addressed in an earlier chapter should be included in this final chapter. The following section headings are suggestions only.>

4.1 Conclusions

Physically based shading produces more realistic frames as it models physical phenomena such as the Fresnel effect and multiple incident lights more accurately than Blinn-Phong shading.

4.2 Future Work

- Using MERL for comparisons (how does industry do it)?
- Comparing the material side of things?
- Compare with indirect lighting effects
- Talk about comparing different PBS models

Bibliography

[1] Mark Claypool, Kajal Claypool, and Feissal Damaa. "The effects of frame rate and resolution on users playing first person shooter games". In: *Multimedia Computing and Networking 2006*. Ed. by Surendar Chandra and Carsten Griwodz. Vol. 6071. International Society for Optics and Photonics. SPIE, 2006, p. 8. DOI: 10.1117/12.648609. URL: https://doi.org/10.1117/12.648609.

[2] Per H. Christensen et al. "Ray Tracing for the Movie 'Cars'". In: *2006 IEEE Symposium on Interactive Ray Tracing*. 2006, p. 5. DOI: 10.1109/RT.2006.280208.

[3] NVIDIA. *NVIDIA Turing GPU Architecture*. Tech. rep. 2018, p. 31.

[4] Johannes Deligiannis and Jan Schmid. *It Just Works: Ray-Traced Reflections in 'Battlefield V'*. EA DICE and GDC. Mar. 18, 2018. URL: https://www.gdcvault.com/play/1026282/It-Just-Works-Ray-Traced (visited on 03/13/2022).

[5] Unity Technologies. *Specular*. URL: https://docs.unity3d.com/560/Documentation/Manual/shader-NormalSpecular.html (visited on 03/15/2022).

[6] Unreal Engine. *Materials Overview*. URL: https://docs.unrealengine.com/udk/Three/MaterialsOverview.html (visited on 03/15/2022).

[7] Khronos. *Fixed Function Pipeline*. URL: https://www.khronos.org/opengl/wiki/Fixed_Function_Pipeline (visited on 03/15/2022).

[8] Bui Tuong Phong. "Illumination for Computer Generated Pictures". In: *Commun. ACM* 18.6 (1975), 311–317. ISSN: 0001-0782. DOI: 10.1145/360825.360839. URL: https://doi.org/10.1145/360825.360839.

[9] B Karis. "Real Shading in Unreal Engine 4". In: *SIGGRAPH 2013 Course: Physically Based Shading in Theory and Practice*. Anaheim, California, 2013, p. 2.

[10] S Lagarde and C Rousiers. "Moving Frostbite to Physically Based Rendering 3.0". In: *SIGGRAPH 2014 Course: Physically Based Shading in Theory and Practice*. Vancouver, 2014.

[11] Stephen Hill et al. "Physically based shading in theory and practice". In: *ACM SIGGRAPH 2020 Courses*. 2020, p. 2.

[12] Brent Burley and Walt Disney Animation Studios. "Physically-based shading at disney". In: *ACM SIGGRAPH*. Vol. 2012. vol. 2012. 2012, p. 18.

[13] Johann Heinrich Lambert and David L DiLaura. *Photometry or On The Measure and Gradations of light, Color, and Shade*. A translation of Lambert's Photometria, which was originally published in 1760. Illuminating Engineering Society of North America, 2001. ISBN: 978-0-87995-179-5.

[14] P.S. Strauss. "A realistic lighting model for computer animators". In: *IEEE Computer Graphics and Applications* 10.6 (1990), pp. 56–64. DOI: 10.1109/38.62696.

[15] Tomas Akenine-Möller et al. *Real-Time Rendering 4th Edition*. Boca Raton, FL, USA: A K Peters/CRC Press, 2018, p. 1200. ISBN: 978-1-13862-700-0.

[16] James F. Blinn. "Models of Light Reflection for Computer Synthesized Pictures". In: *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '77. San Jose, California: Association for Computing Machinery, 1977, 192–198. ISBN: 9781450373555. DOI: 10.1145/563858.563893. URL: https://doi.org/10.1145/563858.563893.

[17] David Murray, Alban Fichet, and Romain Pacanowski. "Efficient Spectral Rendering on the GPU for Predictive Rendering". In: *Ray Tracing Gems II*. Springer, 2021, pp. 673 –698. DOI: 10.1007/978-1-4842-7185-8_42. URL: https://hal.inria.fr/hal-03331619.

[18] M.C. Stone. "Representing colors as three numbers [color graphics]". In: *IEEE Computer Graphics and Applications* 25.4 (2005), pp. 78–85. DOI: 10.1109/MCG.2005.84.

[19] Raju Datla and Albert Parr. "1. Introduction to Optical Radiometry". In: *Experimental Methods in the Physical Sciences* 41 (Dec. 2005), pp. 1–34. DOI: 10.1016/S1079-4042(05)41001-2.

[20] Victor frederick Weisskopf. "How Light Interacts with Matter". In: *Scientific American* 219 (1968), pp. 60–71.

[21] Naty Hoffman. "Background: physics and math of shading". In: *Physically Based Shading in Theory and Practice* 24.3 (2013), pp. 211–223.

[22] Bram van Ginneken, Marigo Stavridi, and Jan J. Koenderink. "Diffuse and Specular Reflectance from Rough Surfaces". In: *Appl. Opt.* 37.1 (1998), pp. 130–139. DOI: 10.1364/AO.37.000130. URL: http://opg.optica.org/ao/abstract.cfm?URI=ao-37-1-130.

[23] Wikipedia contributors. *Snell's law — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-March-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Snell%27s_law&oldid=1074492276.

[24] Peter Shirley et al. "A Practitioners' Assessment of Light Reflection Models". In: Sept. 1997. ISBN: 0-8186-8028-8. DOI: 10.1109/PCCGA.1997.626170.

[25] Nicolas Holzschuch and Romain Pacanowski. "A Two-Scale Microfacet Reflectance Model Combining Reflection and Diffraction". In: *ACM Transactions on Graphics* 36.4 (July 2017). Article 66, p. 12. DOI: 10.1145/3072959.3073621. URL: https://hal.inria.fr/hal-01515948.

[26] Gustav Kortüm. *Reflectance spectroscopy: principles, methods, applications*. Springer Science & Business Media, 2012, p. 3.

[27] Pat Hanrahan and Wolfgang Krueger. "Reflection from Layered Surfaces Due to Subsurface Scattering". In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '93. Anaheim, CA: Association for Computing Machinery, 1993, 165–174. ISBN: 0897916018. DOI: 10.1145/166117.166139. URL: https://doi.org/10.1145/166117.166139.

[28] Jorge Jimenez et al. "Separable Subsurface Scattering". In: *Computer Graphics Forum* (2015).

[29] Fred E Nicodemus et al. “Geometrical considerations and nomenclature for reflectance”. In: *National Bureau of Standards (US) monograph* 160 (1977).

[30] Henrik Wann Jensen et al. “A Practical Model for Subsurface Light Transport”. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’01. New York, NY, USA: Association for Computing Machinery, 2001, 511–518. ISBN: 158113374X. DOI: 10.1145/383259.383319. URL: https://doi.org/10.1145/383259.383319.

[31] Bruce Hapke. *Theory of Reflectance and Emittance Spectroscopy*. 2nd ed. Cambridge University Press, 2012, pp. 264–265. DOI: 10.1017/CBO9781139025683.

[32] Gregory J. Ward. “Measuring and Modeling Anisotropic Reflection”. In: *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’92. New York, NY, USA: Association for Computing Machinery, 1992, 265–272. ISBN: 0897914791. DOI: 10.1145/133994.134078. URL: https://doi.org/10.1145/133994.134078.

[33] Franz Faul. “The influence of Fresnel effects on gloss perception”. In: *Journal of Vision* 19.13 (Nov. 2019). ISSN: 1534-7362. DOI: 10.1167/19.13.1. eprint: https://arvojournals.org/arvo/content_public/journal/jov/938255/i1534-7362-19-13-1.pdf. URL: https://doi.org/10.1167/19.13.1.

[34] Christophe Schlick. “An Inexpensive BRDF Model for Physically-based Rendering”. In: *Computer Graphics Forum* 13.3 (1994), pp. 233–246. DOI: https://doi.org/10.1111/1467-8659.1330233. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1467-8659.1330233. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.1330233.

[35] Addy Ngan, Frédo Durand, and Wojciech Matusik. “Experimental Analysis of BRDF Models”. In: *Eurographics Symposium on Rendering (2005)*. Ed. by Kavita Bala and Philip Dutre. The Eurographics Association, 2005. ISBN: 3-905673-23-1. DOI: 10.2312/EGWR/EGSR05/117-126.

[36] Eric Heitz. “Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs”. In: *Journal of Computer Graphics Techniques (JCGT)* 3.2 (2014), pp. 48–107. ISSN: 2331-7418. URL: http://jcgt.org/published/0003/02/03/.

[37] Eric Heitz et al. “Multiple-Scattering Microfacet BSDFs with the Smith Model”. In: *ACM Trans. Graph.* 35.4 (2016). ISSN: 0730-0301. DOI: 10.1145/2897824.2925943. URL: https://doi.org/10.1145/2897824.2925943.

[38] R. L. Cook and K. E. Torrance. “A Reflectance Model for Computer Graphics”. In: *ACM Trans. Graph.* 1.1 (1982), 7–24. ISSN: 0730-0301. DOI: 10.1145/357290.357293. URL: https://doi.org/10.1145/357290.357293.

[39] Michael Oren and Shree K. Nayar. “Generalization of Lambert’s Reflectance Model”. In: *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’94. New York, NY, USA: Association for Computing Machinery, 1994, 239–246. ISBN: 0897916670. DOI: 10.1145/192161.192213. URL: https://doi.org/10.1145/192161.192213.

[40] S Lagarde. *Spherical Gaussian approximation for Blinn-Phong, Phong and Fresnel*. June 3, 2012. URL: https://seblagarde.wordpress.com/2012/06/03/spherical-gaussien-approximation-for-blinn-phong-phong-and-fresnel (visited on 04/06/2022).

[41] P. Beckmann, J.M. Coulson, and A. Spizzichino. *The Scattering of Electromagnetic Waves from Rough Surfaces*. A Pergamon Press book. Pergamon Press; [distributed in the Western Hemisphere by Macmillan, New York], 1963. ISBN: 9780080100074. URL: https://books.google.co.uk/books?id=QBEIAQAAIAAJ.

[42] Bruce Walter et al. "Microfacet Models for Refraction through Rough Surfaces". In: *Proceedings of the 18th Eurographics Conference on Rendering Techniques*. EGSR'07. Grenoble, France: Eurographics Association, 2007, 195–206. ISBN: 9783905673524.

[43] T. S. Trowbridge and K. P. Reitz. "Average irregularity representation of a rough surface for ray reflection". In: *J. Opt. Soc. Am.* 65.5 (1975), pp. 531–536. DOI: 10.1364/JOSA.65.000531. URL: http://opg.optica.org/abstract.cfm?URI=josa-65-5-531.

[44] K. E. Torrance and E. M. Sparrow. "Theory for Off-Specular Reflection From Roughened Surfaces*". In: *J. Opt. Soc. Am.* 57.9 (1967), pp. 1105–1114. DOI: 10.1364/JOSA.57.001105. URL: http://opg.optica.org/abstract.cfm?URI=josa-57-9-1105.

[45] B. Smith. "Geometrical shadowing of a random rough surface". In: *IEEE Transactions on Antennas and Propagation* 15.5 (1967), pp. 668–671. DOI: 10.1109/TAP.1967.1138991.

[46] Peter S. Shirley. "Physically Based Lighting Calculations for Computer Graphics". UMI Order NO. GAX91-24487. PhD thesis. USA, 1991, pp. 47–48.

[47] Earl Hammon Jr. "PBR Diffuse Lighting for GGX+Smith Microsurfaces". Game Developers Conference (GDC). 2017. URL: https://www.gdcvault.com/play/1024478/PBR-Diffuse-Lighting-for-GGX (visited on 04/08/2022).

[48] Yoshiharu Gotanda. "Designing Reflectance Models for New Consoles". In: 2014.

[49] Dario Seyb et al. "The Design and Evolution of the UberBake Light Baking System". In: *ACM Trans. Graph.* 39.4 (2020). ISSN: 0730-0301. DOI: 10.1145/3386569.3392394. URL: https://doi.org/10.1145/3386569.3392394.

[50] Ravi Ramamoorthi and Pat Hanrahan. "An Efficient Representation for Irradiance Environment Maps". In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. New York, NY, USA: Association for Computing Machinery, 2001, 497–500. ISBN: 158113374X. DOI: 10.1145/383259.383317. URL: https://doi.org/10.1145/383259.383317.

[51] Erik Reinhard et al. "Photographic Tone Reproduction for Digital Images". In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '02. San Antonio, Texas: Association for Computing Machinery, 2002, 267–276. ISBN: 1581135211. DOI: 10.1145/566570.566575. URL: https://doi.org/10.1145/566570.566575.

[52] Naty Hoffman. "Outside the Echo Chamber: Learning from Other Disciplines, Industries, and Art Forms". ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. 2013. URL: http://renderwonk.com/publications/i3d2013-keynote/i3dkeynote-final.pdf (visited on 04/10/2022).

[53] John Hable. "Uncharted 2: HDR Lighting". Game Developers Conference (GDC). 2010. URL: https://www.gdcvault.com/play/1012351/Uncharted-2-HDR (visited on 04/10/2022).

[54] Unreal Engine. *Color Grading and Filmic Tonemapper*. URL: https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/PostProcessEffects/ColorGrading/ (visited on 04/10/2022).

[55] Larry Gritz and Eugene d'Eon. *Gpu Gems 3: Chapter 24. The Importance of Being Linear*. First. Addison-Wesley Professional, 2007. ISBN: 9780321545428. URL: https://developer.nvidia.com/gpugems/gpugems3/part-iv-image-effects/chapter-24-importance-being-linear.

[56] Kurt Akeley. "Reality Engine Graphics". In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '93. Anaheim, CA: Association for Computing Machinery, 1993, 109–116. ISBN: 0897916018. DOI: 10.1145/166117.166131. URL: https://doi.org/10.1145/166117.166131.

[57] James F. Blinn. "Simulation of Wrinkled Surfaces". In: *SIGGRAPH Comput. Graph.* 12.3 (1978), 286–292. ISSN: 0097-8930. DOI: 10.1145/965139.507101. URL: https://doi.org/10.1145/965139.507101.

[58] *glTF 2.0 Specification*. 2.0.1. Khronos Group. Nov. 2021, p. 48. URL: https://www.khronos.org/registry/glTF/specs/2.0/glTF-2.0.pdf (visited on 04/15/2022).

[59] S Lagarde. *PI or not to PI in game lighting equation*. Jan. 8, 2012. URL: https://seblagarde.wordpress.com/2012/01/08/pi-or-not-to-pi-in-game-lighting-equation/ (visited on 04/15/2022).

[60] Xim Cerda-Company, C. Alejandro Parraga, and Xavier Otazu. "Which tone-mapping operator is the best? A comparative study of perceptual quality". In: *J. Opt. Soc. Am. A* 35.4 (2018), pp. 626–638. DOI: 10.1364/JOSAA.35.000626. URL: http://opg.optica.org/josaa/abstract.cfm?URI=josaa-35-4-626.

[61] Markus Billeter. *COMP5822M – High Perf. Graphics - Lecture 11: Shading*. 2021/2022.

Appendix A

Self-appraisal

<This appendix should contain everything covered by the 'self-appraisal' criterion in the mark scheme. Although there is no length limit for this section, 2—4 pages will normally be sufficient. The format of this section is not prescribed, but you may like to organise your discussion into the following sections and subsections.>

A.1 Critical self-evaluation

A.2 Personal reflection and lessons learned

A.3 Legal, social, ethical and professional issues

<Refer to each of these issues in turn. If one or more is not relevant to your project, you should still explain *why* you think it was not relevant.>

A.3.1 Legal issues

A.3.2 Social issues

A.3.3 Ethical issues

A.3.4 Professional issues

Appendix B

External Material

<This appendix should provide a brief record of materials used in the solution that are not the student's own work. Such materials might be pieces of codes made available from a research group/company or from the internet, datasets prepared by external users or any preliminary materials/drafts/notes provided by a supervisor. It should be clear what was used as ready-made components and what was developed as part of the project. This appendix should be included even if no external materials were used, in which case a statement to that effect is all that is required.>

Three types of external material were used in my solution: software libraries, an implementation of the ACES tone mapping operator, and 3D Models.

B.1 Software Libraries

The libraries used in my renderer are listed below. For each one, a description is provided, along with a link. Note that the separation between my code and the external libraries is very clear in the code base, with all libraries being present under the `Application/Vendor` directory.

Library Name	**Description**	**Link**
GLFW	A cross-platform utility library that provides windowing, OpenGL contexts and retrieves input events	`https://www.glfw.org/`
Glad	A library for loading pointers to the OpenGL functions	`https://glad.dav1d.de/`
GLM	A maths library specifically designed for use with OpenGL	`https://github.com/g-truc/glm`
spdlog	A fast logging library	`https://github.com/gabime/spdlog`
stb_image	An image loading library that is used when creating textures	`https://github.com/nothings/stb/blob/master/stb_image.h`
assimp	A library to import 3D models of various different file types	`https://github.com/assimp/assimp`

B.2 ACES Tone Mapping Operator

The implementation of the ACES tone mapping operator, which is given in Listing 11, is a slightly modified version of Stephen Hill's code. His code can be accessed via the following link: `https://github.com/TheRealMJP/BakingLab/blob/master/BakingLab/ACES.hlsl`

B.3 3D Models

Appendix C

Mathematical Notation

$\mathbf{n}$	Normal vector
$\mathbf{l}$	Light direction
$\mathbf{v}$	View vector
$\mathbf{h}$	Half vector
$\mathbf{a} \cdot \mathbf{b}$	The dot product of vectors $\mathbf{a}$ and $\mathbf{b}$
$\|\mathbf{a}\|$	The norm of vector $\mathbf{a}$
$\|x\|$	The absolute value of x
x^+	Clamp x to 0 if $x < 0$
$\mathcal{X}^+(x)$	Returns 1 if $x > 0$, else returns 0
$lerp(x, y, t)$	Linearly interpolates between x and y by the interpolant t: $lerp(x, y, t) = x(1 - t) + yt$

Appendix D

Shader Code

All the shader code can be accessed in the code base under the `Application/Assets/Shaders` directory. The code is also given below as it is useful to be able to view the end-to-end process of shading. As is demanded by most graphics APIs, each shader program given below is split into a vertex and fragment shader.

D.1 Blinn-Phong Shader

D.1.1 Vertex Shader

```
#version 460 core

// ATTRIBUTES

layout (location = 0) in vec3 a_position;
layout (location = 1) in vec3 a_normal;
layout (location = 2) in vec2 a_textureCoordinates;
layout (location = 3) in vec3 a_tangent;
layout (location = 4) in vec3 a_bitangent;

// UNIFORMS

uniform mat4 u_transform;
uniform mat4 u_projectionViewMatrix;

// OUTPUTS

struct VertexOutput
{
        vec3 worldPosition;
        vec3 normal;
        vec2 textureCoordinates;
        mat3 TBN;
};

out VertexOutput vertex_output;

// FUNCTIONS

void main()
{
        vertex_output.worldPosition = vec3(u_transform * vec4(a_position, 1.0f));
        vertex_output.normal = normalize(vec3(transpose(inverse(u_transform)) * vec4(a_normal,
        ↪  0.0f)));
        vertex_output.textureCoordinates = a_textureCoordinates;

        vec3 normalTransformed = normalize(vec3(u_transform * vec4(a_normal, 0.0f)));
```

```
37          vec3 tangentTransformed = normalize(vec3(u_transform * vec4(a_tangent, 0.0f)));
38          vec3 bitangentTransformed = normalize(vec3(u_transform * vec4(a_bitangent, 0.0f)));
39          vertex_output.TBN = mat3(tangentTransformed, bitangentTransformed, normalTransformed);
40
41          gl_Position = u_projectionViewMatrix * u_transform * vec4(a_position, 1.0f);
42  }
```

D.1.2 Fragment Shader

```
1   #version 460 core
2
3   // INPUTS FROM VERTEX SHADER
4
5   struct VertexOutput
6   {
7           vec3 worldPosition;
8           vec3 normal;
9           vec2 textureCoordinates;
10          mat3 TBN;
11  };
12
13  in VertexOutput vertex_output;
14
15  // UNIFORMS
16
17  // Material
18
19  struct Material
20  {
21          vec4 diffuseColor;
22          vec3 specularColor;
23          sampler2D diffuseMap;
24          sampler2D specularMap;
25          sampler2D normalMap;
26          float shininess;
27
28          bool useNormalMap;
29  };
30
31  uniform Material u_material;
32
33  // Point lights
34
35  struct PointLight
36  {
37          vec3 worldPosition;
38          vec3 diffuseComponent;
39          vec3 specularComponent;
40          float lightRadius;
41  };
42
43  const uint MAX_NUMBER_OF_POINT_LIGHTS = 128;
```

```
44 uniform PointLight u_pointLights[MAX_NUMBER_OF_POINT_LIGHTS];
45 uniform uint u_pointLightNumber;
46
47 uniform vec3 u_viewPosition;
48
49 // OUTPUTS
50
51 layout(location = 0) out vec4 o_fragColor;
52
53 // CONSTANTS
54
55 const vec3 LIGHT_AMBIENT = vec3(0.02f);
56
57 // GLOBAL DATA
58
59 vec3 g_normal;
60 vec3 g_viewDirection;
61 vec3 g_diffuseMaterialValue;
62 vec3 g_specularMaterialValue;
63
64 // FUNCTIONS
65
66 vec3 getNormalisedSurfaceNormal()
67 {
68         if (u_material.useNormalMap)
69         {
70                 vec4 sampleFromNormalMap = texture(u_material.normalMap,
                  ↪  vertex_output.textureCoordinates);
71                 vec3 sampledNormal = (sampleFromNormalMap.rgb * 2.0f) - 1.0f;
72                 vec3 sampledNormalInWorldSpace = vertex_output.TBN * sampledNormal;
73                 return normalize(sampledNormalInWorldSpace);
74         }
75         else
76         return normalize(vertex_output.normal);
77 }
78
79 float calculatePointLightAttenuationFactor(float lightDistance, float lightRadius)
80 {
81         const float lightSize = 0.01f;
82         const uint n = 4;
83
84         // Restrict the minimum value of the denominator to 0.01 * 0.01 to avoid the value
85         // exploding or having divide by zero errors
86         float inverseSquaredDistance = 1.0f / pow(max(lightDistance, lightSize), 2.0f);
87
88         // Use a windowing function to cutoff the attenuation value to 0 at large distances
89
90         float lightDistanceNOverLightRadiusN = 1.0f - pow(lightDistance / lightRadius, n);
91         float windowingFunctionValue = pow(clamp(lightDistanceNOverLightRadiusN, 0.0f, 1.0f),
          ↪  2.0f);
92
93         return min(inverseSquaredDistance * windowingFunctionValue, 1.0f);
94 }
95
```

```
96  vec3 calculateDiffuseContribution(vec3 lightDirection, vec3 attenuatedLightDiffuseComponent)
97  {
98          float lightIncidentDiffuseFactor = max(dot(lightDirection, g_normal), 0.0f);
99          return (g_diffuseMaterialValue * lightIncidentDiffuseFactor) *
            ↪  attenuatedLightDiffuseComponent;
100 }
101
102 vec3 calculateSpecularContribution(vec3 lightDirection, vec3 attenuatedLightSpecularComponent)
103 {
104         vec3 halfVector = normalize(g_viewDirection + lightDirection);
105         float lightReflectedSpecularFactor = pow(max(dot(halfVector, g_normal), 0.0f),
            ↪  u_material.shininess);
106         return (g_specularMaterialValue * lightReflectedSpecularFactor) *
            ↪  attenuatedLightSpecularComponent;
107 }
108
109 vec3 gammaCorrectColor(vec3 color)
110 {
111         vec3 SRGBEncodedHigher = (1.055f * pow(color, vec3(1.0f / 2.4f))) - 0.055f;
112         vec3 SRGBEncodedLower = 12.92f * color;
113         float rSRGBEncoded = (color.r > 0.0031308f) ? SRGBEncodedHigher.r : SRGBEncodedLower.r;
114         float gSRGBEncoded = (color.g > 0.0031308f) ? SRGBEncodedHigher.g : SRGBEncodedLower.g;
115         float bSRGBEncoded = (color.b > 0.0031308f) ? SRGBEncodedHigher.b : SRGBEncodedLower.b;
116         return vec3(rSRGBEncoded, gSRGBEncoded, bSRGBEncoded);
117 }
118
119 void main()
120 {
121         vec3 color;
122         g_normal = getNormalisedSurfaceNormal();
123         g_viewDirection = normalize(u_viewPosition - vertex_output.worldPosition);
124
125         vec4 diffuseMaterialValueWithAlpha = texture(u_material.diffuseMap,
            ↪  vertex_output.textureCoordinates) * u_material.diffuseColor;
126         float alpha = diffuseMaterialValueWithAlpha.a;
127
128         g_diffuseMaterialValue = diffuseMaterialValueWithAlpha.rgb;
129         g_specularMaterialValue = texture(u_material.specularMap,
            ↪  vertex_output.textureCoordinates).rgb * u_material.specularColor;
130
131         // Calculate ambient contribution
132
133         color = g_diffuseMaterialValue * LIGHT_AMBIENT;
134
135         // Calculate diffuse and specular contribution for each light
136
137         for (uint i = 0; i < u_pointLightNumber; i++)
138         {
139                 PointLight pointLight = u_pointLights[i];
140                 vec3 lightDirection = normalize(pointLight.worldPosition -
                    ↪  vertex_output.worldPosition);
141
142                 float lightDistance = length(pointLight.worldPosition -
                    ↪  vertex_output.worldPosition);
```

```
143                 float attenuationFactor = calculatePointLightAttenuationFactor(lightDistance,
                    ↪  pointLight.lightRadius);
144
145                 color += calculateDiffuseContribution(lightDirection,
                    ↪  pointLight.diffuseComponent * attenuationFactor);
146                 color += calculateSpecularContribution(lightDirection,
                    ↪  pointLight.specularComponent * attenuationFactor);
147         }
148
149         o_fragColor = vec4(gammaCorrectColor(color), alpha);
150 }
```

D.2 Physically Based Shaders

D.2.1 PBS Vertex Shader

```
1  #version 460 core
2
3  // ATTRIBUTES
4
5  layout (location = 0) in vec3 a_position;
6  layout (location = 1) in vec3 a_normal;
7  layout (location = 2) in vec2 a_textureCoordinates;
8  layout (location = 3) in vec3 a_tangent;
9  layout (location = 4) in vec3 a_bitangent;
10
11 // UNIFORMS
12
13 uniform mat4 u_transform;
14 uniform mat4 u_projectionViewMatrix;
15
16 // OUTPUTS
17
18 struct VertexOutput
19 {
20         vec3 worldPosition;
21         vec3 normal;
22         vec2 textureCoordinates;
23         mat3 TBN;
24 };
25
26 out VertexOutput vertex_output;
27
28 // FUNCTIONS
29
30 void main()
31 {
32         vertex_output.worldPosition = vec3(u_transform * vec4(a_position, 1.0f));
33         vertex_output.normal = normalize(vec3(transpose(inverse(u_transform)) * vec4(a_normal,
           ↪  0.0f)));
34         vertex_output.textureCoordinates = a_textureCoordinates;
35
```

```
36          vec3 normalTransformed = normalize(vec3(u_transform * vec4(a_normal, 0.0f)));
37          vec3 tangentTransformed = normalize(vec3(u_transform * vec4(a_tangent, 0.0f)));
38          vec3 bitangentTransformed = normalize(vec3(u_transform * vec4(a_bitangent, 0.0f)));
39          vertex_output.TBN = mat3(tangentTransformed, bitangentTransformed, normalTransformed);
40
41          gl_Position = u_projectionViewMatrix * u_transform * vec4(a_position, 1.0f);
42  }
```

D.2.2 PBS Fragment Shader

```
1   #version 460 core
2
3   // INPUTS FROM VERTEX SHADER
4
5   struct VertexOutput
6   {
7           vec3 worldPosition;
8           vec3 normal;
9           vec2 textureCoordinates;
10          mat3 TBN;
11  };
12
13  in VertexOutput vertex_output;
14
15  // UNIFORMS
16
17  // Material
18
19  struct Material
20  {
21          vec4 baseColor;
22          float roughness;
23          float metalness;
24          sampler2D baseColorMap;
25          sampler2D roughnessMap;
26          sampler2D metalnessMap;
27          sampler2D normalMap;
28
29          bool useNormalMap;
30  };
31
32  uniform Material u_material;
33
34  // Point lights
35
36  struct PointLight
37  {
38          vec3 worldPosition;
39          vec3 lightColor;
40          float luminousPower;
41          float lightRadius;
42  };
```

```
43
44  const uint MAX_NUMBER_OF_POINT_LIGHTS = 128;
45  uniform PointLight u_pointLights[MAX_NUMBER_OF_POINT_LIGHTS];
46  uniform uint u_pointLightNumber;
47
48  uniform vec3 u_viewPosition;
49
50  uniform float u_exposure;
51
52  // OUTPUTS
53
54  layout(location = 0) out vec4 o_fragColor;
55
56  // CONSTANTS
57
58  const vec3 F0_FOR_DIELECTRICS = vec3(0.04f);
59  const float PI = 3.14159265359;
60
61  // GLOBAL DATA
62
63  struct Directions
64  {
65          vec3 normal;
66          vec3 viewDirection;
67  };
68
69  Directions g_directions;
70
71  struct DotProducts
72  {
73          float nDotV;
74  };
75
76  DotProducts g_dotProducts;
77
78  struct MaterialProperties
79  {
80          vec3 baseColor;
81          float alpha;
82          float roughness;
83          float metalness;
84
85          vec3 f0;
86  };
87
88  MaterialProperties g_materialProperties;
89
90  // FUNCTIONS
91
92  /*
93  Used the Sclick approximation to calculate the Fresnel reflectance
94  */
95  vec3 calculateFresnelSchlickApproximation(vec3 f0, float u)
96  {
```

```
97          return f0 + (1 - f0) * pow((1 - u), 5.0f);
98  }
99
100 /*
101 Used the Smith height-correlated masking-shadowing function.
102
103 As pointed out by Lagarde, using a combination of Smith and the GGX NDF
104 in the specular (surface reflection) BRDF term means optimisations can be made.
105
106 Namely, G(l,v) / (4 * |n.l| * |n.v|) can be simplified. Hammon gives an accurate
107 approximation for the above term. This approximation is being calculated in the function.
108
109 See https://www.gdcvault.com/play/1024478/PBR-Diffuse-Lighting-for-GGX
110 */
111 float calculateHammonSmithMaskingSpecularDenominatorAppoximation(float nDotL)
112 {
113         float alpha = g_materialProperties.roughness * g_materialProperties.roughness;
114         float x = 2.0f * abs(nDotL) * abs(g_dotProducts.nDotV);
115         float y = abs(nDotL) + abs(g_dotProducts.nDotV);
116         return 1.0f / (2.0f * mix(x, y, alpha));
117 }
118
119 /*
120 Use the GGX (Trowbridge-Reitz) distribution for the NDF.
121
122 Also used the Disney mapping of alpha = roughness * roughness, where roughness
123 then gives a perceptually linear change from [0, 1].
124 */
125 float calculateGGXDistribution(float nDotH)
126 {
127         float alpha = g_materialProperties.roughness * g_materialProperties.roughness;
128         float alpha2 = alpha * alpha;
129         float x = 1 + (nDotH * nDotH * (alpha2 - 1.0f));
130         return alpha2 / (PI * x * x);
131 }
132
133 vec3 getNormalisedSurfaceNormal()
134 {
135         if (u_material.useNormalMap)
136         {
137                 vec4 sampleFromNormalMap = texture(u_material.normalMap,
                        ↪  vertex_output.textureCoordinates);
138                 vec3 sampledNormal = (sampleFromNormalMap.rgb * 2.0f) - 1.0f;
139                 vec3 sampledNormalInWorldSpace = vertex_output.TBN * sampledNormal;
140                 return normalize(sampledNormalInWorldSpace);
141         }
142         else
143         return normalize(vertex_output.normal);
144 }
145
146 float calculatePointLightAttenuationFactor(float lightDistance, float lightRadius)
147 {
148         const float lightSize = 0.01f;
149         const uint n = 4;
```

```
150
151         // Restrict the minimum value of the denominator to 0.01 * 0.01 to avoid the value
152         // exploding or having divide by zero errors
153         float inverseSquaredDistance = 1.0f / pow(max(lightDistance, lightSize), 2.0f);
154
155         // Use a windowing function to cutoff the attenuation value to 0 at large distances
156
157         float lightDistanceNOverLightRadiusN = 1.0f - pow(lightDistance / lightRadius, n);
158         float windowingFunctionValue = pow(clamp(lightDistanceNOverLightRadiusN, 0.0f, 1.0f),
            ↪  2.0f);
159
160         return min(inverseSquaredDistance * windowingFunctionValue, 1.0f);
161 }
162
163 vec3 calculatePointLightContribution(const PointLight pointLight)
164 {
165         // Calculate the incoming radiance from the point light
166
167         float lightDistance = length(pointLight.worldPosition - vertex_output.worldPosition);
168         float lightAttenuationFactor = calculatePointLightAttenuationFactor(lightDistance,
            ↪  pointLight.lightRadius);
169
170         float luminousIntensity = pointLight.luminousPower / (4.0f * PI);
171         vec3 lightRadiance = pointLight.lightColor * luminousIntensity * lightAttenuationFactor;
172
173         // Initialise values
174
175         vec3 lightDirection = normalize(pointLight.worldPosition - vertex_output.worldPosition);
176         vec3 halfVector = normalize(g_directions.viewDirection + lightDirection);
177
178         float nDotL = dot(g_directions.normal, lightDirection);
179         float hDotL = dot(halfVector, lightDirection);
180         float nDotH = dot(g_directions.normal, halfVector);
181
182         // Specular (surface reflection) term
183
184         // fresnelReflectance is also the specularTermContribution
185         vec3 fresnelReflectance = calculateFresnelSchlickApproximation(g_materialProperties.f0,
            ↪  max(hDotL, 0.0f));
186         float hammonSmithMaskingSpecularDenominatorApproximation =
            ↪  calculateHammonSmithMaskingSpecularDenominatorAppoximation(nDotL);
187         float NDF = calculateGGXDistribution(nDotH);
188
189         vec3 specularTerm = fresnelReflectance *
            ↪  hammonSmithMaskingSpecularDenominatorApproximation * NDF;
190
191         // Diffuse (sub-surface reflection) term
192
193         vec3 diffuseTermContribution = (vec3(1.0f) - fresnelReflectance) * (1.0f -
            ↪  g_materialProperties.metalness);
194         vec3 diffuseTerm = diffuseTermContribution * (g_materialProperties.baseColor / PI);
195
196         // Integrate the reflectance equation with respect to this light
197
```

```
198         vec3 BRDFValue = diffuseTerm + specularTerm;
199         return BRDFValue * lightRadiance * max(nDotL, 0.0f);
200 }
201
202 void main()
203 {
204         // Initialise global values
205
206         g_directions.normal = getNormalisedSurfaceNormal();
207         g_directions.viewDirection = normalize(u_viewPosition - vertex_output.worldPosition);
208
209         g_dotProducts.nDotV = dot(g_directions.normal, g_directions.viewDirection);
210
211         vec4 baseColorWithAlpha = texture(u_material.baseColorMap,
            ↪ vertex_output.textureCoordinates) * u_material.baseColor;
212         g_materialProperties.baseColor = baseColorWithAlpha.rgb;
213         g_materialProperties.alpha = baseColorWithAlpha.a;
214         g_materialProperties.roughness = texture(u_material.roughnessMap,
            ↪ vertex_output.textureCoordinates).r * u_material.roughness;
215         g_materialProperties.metalness = texture(u_material.metalnessMap,
            ↪ vertex_output.textureCoordinates).r * u_material.metalness;
216         g_materialProperties.f0 = mix(F0_FOR_DIELECTRICS, g_materialProperties.baseColor,
            ↪ g_materialProperties.metalness);
217
218         // Solve the reflectance equation by evaluating the contribution of each point light
219
220         vec3 fragmentColor = vec3(0.0f);
221
222         for (uint i = 0; i < u_pointLightNumber; i++)
223         fragmentColor += calculatePointLightContribution(u_pointLights[i]);
224
225         // Apply a rudimentary ambient term
226
227         vec3 ambientTerm = vec3(0.05f) * g_materialProperties.baseColor;
228         fragmentColor += ambientTerm;
229
230         // Output the shaded color
231
232         o_fragColor = vec4(fragmentColor, g_materialProperties.alpha);
233 }
```

D.2.3 Post Processing Vertex Shader

```
1 #version 460 core
2
3 // ATTRIBUTES
4
5 layout (location = 0) in vec2 a_position;
6 layout (location = 1) in vec2 a_textureCoordinates;
7
8 // OUTPUTS
9
```

```
10 struct VertexOutput
11 {
12         vec2 textureCoordinates;
13 };
14
15 out VertexOutput vertex_output;
16
17 // FUNCTIONS
18
19 void main()
20 {
21         vertex_output.textureCoordinates = a_textureCoordinates;
22         gl_Position = vec4(a_position, 0.0f, 1.0f);
23 }
```

D.2.4 Post Processing Fragment Shader

```
1  #version 460 core
2
3  // INPUTS FROM VERTEX SHADER
4
5  struct VertexOutput
6  {
7          vec2 textureCoordinates;
8  };
9
10 in VertexOutput vertex_output;
11
12 // UNIFORMS
13
14 uniform sampler2D u_inputTexture;
15 uniform float u_exposure;
16
17 // OUTPUTS
18
19 layout(location = 0) out vec4 o_fragColor;
20
21 // FUNCTIONS
22
23 /*
24 ACES Tone Mapping
25
26 Curve adapted from: https://github.com/TheRealMJP/BakingLab/blob/master/BakingLab/ACES.hlsl
27 */
28 vec3 applyToneMapping(vec3 color)
29 {
30         mat3 inputMatrix = mat3
31         (
32         0.59719f,  0.07600f,  0.02840f,
33         0.35458f,  0.90834f,  0.13383f,
34         0.04823f,  0.01566f,  0.83777f
35         );
```

```
36
37          mat3 outputMatrix = mat3
38          (
39          1.60475f, -0.10208f, -0.00327f,
40          -0.53108f,  1.10813f, -0.07276f,
41          -0.07367f, -0.00605f,  1.07602f
42          );
43
44          color = inputMatrix * color;
45          vec3 a = color * (color + 0.0245786f) - 0.000090537f;
46          vec3 b = color * (0.983729f * color + 0.4329510f) + 0.238081f;
47          color = a / b;
48
49          return clamp(outputMatrix * color, 0.0f, 1.0f);
50  }
51
52  vec3 gammaCorrectColor(vec3 color)
53  {
54          vec3 SRGBEncodedHigher = (1.055f * pow(color, vec3(1.0f / 2.4f))) - 0.055f;
55          vec3 SRGBEncodedLower = 12.92f * color;
56          float rSRGBEncoded = (color.r > 0.0031308f) ? SRGBEncodedHigher.r : SRGBEncodedLower.r;
57          float gSRGBEncoded = (color.g > 0.0031308f) ? SRGBEncodedHigher.g : SRGBEncodedLower.g;
58          float bSRGBEncoded = (color.b > 0.0031308f) ? SRGBEncodedHigher.b : SRGBEncodedLower.b;
59          return vec3(rSRGBEncoded, gSRGBEncoded, bSRGBEncoded);
60  }
61
62  void main()
63  {
64          vec4 inputColor = texture(u_inputTexture, vertex_output.textureCoordinates);
65          float alpha = inputColor.a;
66
67          // Apply exposure, tone mapping and gamma correction
68
69          inputColor *= u_exposure;
70
71          o_fragColor = vec4(gammaCorrectColor(applyToneMapping(inputColor.rgb)), alpha);
72  }
```