BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

MATERIAL VISUALIZATION USING GLSL SHADERS

VIZUALIZACE MATERIÁLŮ V GLSL SHADERECH

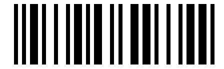
BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **AUTOR PRÁCE**	**MARTIN MARŠALEK**
SUPERVISOR **VEDOUCÍ PRÁCE**	**Ing. JAN PEČIVA, Ph.D.**

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment



162916

Institut:	Department of Computer Graphics and Multimedia (DCGM)
Student:	**Maršalek Martin**
Programme:	Information Technology
Title:	**Material visualization using GLSL shaders**
Category:	Computer Graphics
Academic year:	2024/25

Assignment:

1. Get acquainted with glTF format for 3D scenes and models. Study various material models, especially those used by glTF format. Study Vulkan API, especially the parts most related to material rendering.
2. Design algorithms for visualization of material models, especially those used by glTF format.
3. Implement the algorithms. Incorporate your algorithms either to CADR project that is developed on our faculty, or create your Vulkan application for visualization of 3D models in glTF format and incorporate your algorithms into it.
4. Demonstrate the capabilities of your algorithms using set of models showing various material effects. Discuss possible future development and useful extensions of your solution.
5. Publish your work on internet, for example on https://github.com/Vulkan-FIT/. Consider publishing your code under some open-source license.

Literature:

- https://google.github.io/filament/
- https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html
- follow instructions of your supervisor

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Pečiva Jan, Ing., Ph.D.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2024
Submission deadline:	14.5.2025
Approval date:	12.11.2024

Abstract
This thesis concerns the study of material visualization techniques intended specifically for the material model of the glTF format. The result is an application able to load, process and render glTF models including their material model. The application was developed using the Vulkan API and the C++ programming language. PBR algorithms were implemented, namely the Cook-Torrance model with the GGX normal distribution function.

Abstrakt
Táto práca sa zaoberá štúdiou metód zobrazenia materiálového modelu určeného pre formát 3D modelov glTF. Výsledkom je aplikácia schopná načítať, spracovať a zobraziť modely formátu glTF vrátane ich materiálového modelu. Pre tvorbu aplikácie bolo využité rozhranie Vulkan a jazyk C++. Implementované boli zobrazovacie algoritmy PBR, konkrétne model Cook-Torrance s funkciou distribúcie normálov GGX.

Keywords
Vulkan, GLSL, glTF, PBR, Cook-Torrance, GGX

Klíčová slova
Vulkan, GLSL, glTF, PBR, Cook-Torrance, GGX

Reference
MARŠALEK, Martin. *Material visualization using GLSL shaders.* Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jan Pečiva, Ph.D.

Rozšířený abstrakt

Táto práca sa zaoberá tématikou vizualizácie materiálových vlastností, konkrétne materiálového modelu formátu 3D modelov glTF skupiny Khronos. GL Transmission Format, skrátený ako glTF je moderný bezplatný formát pre ukladanie dát týkajúcich sa trojdimenzionálnych scén, určených pre vykresľovanie na rôznych platformách vrátane webových stránok. Hlavným cieľom je čo najjednoduchšie a najefektívnejšie načítanie a manipulovanie s dátami modelu. Formát je definovaný špecifikáciou, konkrétne najaktuálnejšou verziou 2.0, ktorá definuje usporiadanie a štruktúru rôznych vlastností, ktoré sú súčasťou formátu. Dôležité pre túto prácu je materiálový model, založený na koncepte "Physically Based Rendering", alebo PBR. Presnejšie je to model, ktorý vypočíta výslednú farbu fragmentu miešaním dvoch modelov pre dielektrické a kovové materiály. Pomer týchto modelov je určený hodnotou, ktorá je súčasťou materiálového modelu. Okrem tejto hodnoty materiálový model obsahuje vlastnosti bežné pre aplikácie využívajúce PBR ako napríklad herné enginy.

Súčasťou práce bol prieskum histórie osvetľovacích modelov využitých pre vykresľovanie materiálového modelu glTF, konkrétne modelov založených na skutočných fyzikálnych interakciách medzi svetlom a rôznymi povrchmi. Snaha bola klásť dôraz na konkrétne modely relevantné pre prácu, teda model Cook-Torrance, ktorý je založený na teórii mikrofazetov a skladá sa z viacerých funkcií, ako funkcie distribúcie normálov Trowbridge-Reitz/GGX, geometrickej funkcie Smith-GGX a zjednodušenej formy fresnelovej rovnice Fresnel-Schlick. Práca obsahuje taktiež popis materiálového modelu glTF vrátane prehľadu jednotlivých vlastností ako napríklad normálovej mapy, emisnej mapy atď. Ďalej bol vykonaný prieskum histórie použitia PBR v grafických aplikáciách vykresľujúcich v reálnom čase, samotného formátu glTF a rozhrania Vulkan.

Praktickou časťou práce je teda návrh a implementácia grafickej aplikácie umožňujúcej vizualizáciu materiálového modelu definovaného špecifikáciou glTF 2.0. Konkrétna forma aplikácie je zobrazovač modelov glTF schopný načítať, spracovať a vykresliť model tak, ako bol navrhnutý tvorcom. Aplikácia je vytvorená za použitia rozhrania Vulkan, ktoré sa radí medzi moderné rozhrania pre prácu s GPU, umožňujúce vývojárom väčšiu kontrolu nad funkciou grafickej karty. Práca s Vulkanom je ale komplikovanejšia ako so staršími, zaužívanejšími rozhraniami a vyžaduje od vývojára presný manažment zdrojov a synchronizáciu operácii grafickej karty.

Aplikácia je určená pre platformu Windows a je implementovaná za použitia programovacieho jazyka C++ a viacerých knižníc. Pri implementácii bola snaha držať sa moderných techník a pravidiel tvorby C++ aplikácií. Finálnym výsledkom je aplikácia pre zobrazovanie glTF modelov využívajúca predom spomenutý osvetľovací model, ktorá umožňuje užívateľovi načítať a zobraziť model, vrátane grafického užívateľského rozhrania umožňujúceho užívateľovi ovládať kameru a vlastnosti orbitujúcich svetiel.

Material visualization using GLSL shaders

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Jan Pečiva Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Martin Maršalek
May 12, 2025

Acknowledgements

I would like to thank my supervisor, Mr. Ing. Jan Pečiva Ph.D for his guidance, patience and assistance with the thesis.

Contents

1 Introduction **4**

2 Theory **5**

2.1 Physics of light propagation . . . 5
2.2 Models of light propagation for use in rendering . . . 8
2.3 Physically based rendering . . . 14
2.4 Models of material properties . . . 16
2.5 glTF . . . 18
2.6 Vulkan . . . 20

3 Design **21**

3.1 Application design . . . 21
3.2 Loading models . . . 22
3.3 Shading model . . . 22

4 Implementation **23**

4.1 Used technology . . . 23
4.2 Scene graph . . . 24
4.3 Loader . . . 25
4.4 Transferring scene data to the GPU . . . 26
4.5 Rendering process . . . 28
4.6 Lights . . . 28
4.7 Implemented shading model . . . 29
4.8 Results . . . 32

5 Conclusion **36**

Bibliography **37**

List of Figures

2.1 Breakdown of frequencies exhibited by electro-magnetic radiation, with the visible spectrum highlighted as a subset of the „solar spectrum“. [9] 6
2.2 As light impacts the boundary of two media, a portion of it is reflected approximately following the angle of reflection, while another portion enters the medium, bouncing inside of it between atoms before exiting the surface. [1] 6
2.3 The light (l), view (v), normal (n) and halfway vector (h) shown in relation to the point (p) on a two dimensional sufrace. [1] 7
2.4 The Phong shading model is a combination of an ambient, diffuse and specular term, each representing different interactions between light and the rendered object. [1] 9
2.5 A visualization of the surface of an object at a microscopic scale. The microsurface illustrated on the image on the left is less rough than the microsurface on the image on the right. Reflected light rays are scattered in various directions depending on the roughness of the surface. [1] 11
2.6 The top left image shows a round object with very low micro surface roughness, with incident light scattering visualized by the image on the top left. The bottom two images show the same phenomenon except with a higher micro surface roughness. [1] 12
2.7 This image illustrates the difference between the microsurface in black and the flat macrosurface in blue. Two distinct normal vectors, one for the microsurface (m) and one for the macrosurface (n) are shown originating in points on their respecting geometric surfaces. [17] 12
2.8 A visualization of the phenomenon of shadowing, where the reflected vector in the middle never reaches the camera due to being blocked by the microsurface. The vector m represents the microsurface normal, while the vectors i and o represent incoming and outgoing light respectively. [17] 13
2.9 This still from the Disney produced film Wreck-It Ralph, showcasing the results of their new physically based shading workflow. [2] 14
2.10 An example of the glTF material model. The left side shows the geometry of the model, while the right side shows slices of the various textures available as material properties. [5] 17
2.11 glTF model format structure. [5] 19

4.1 Diagram showing the relationships between individual components and libraries forming the application. External libraries are designated by an italicized name. 25

4.2 Diagram of the structures defined and used by the application to store the loaded scene, or model, in memory. 26
4.3 Diagram showing the full process of loading a glTF model file. 27
4.4 Diagram showing a high-level view of the intended glTF metallic-roughness shading model. [5] 31
4.5 Images of the rendered model with the output color set to values sampled from texture material properties. From left to right these are the normal, roughness and metallic texture. The vectors sampled from the normal texture are in tangent space. 32
4.6 Images of the rendered model with the output color set to values sampled from the two remaining texture properties. The are, from left to right, the ambient occlusion texture and the emissive texture. 33
4.7 The left image shows the model rendered using only the Lambertian BRDF without accounting for the cosine factor (mentioned in section 2.2), while the right image shows the model lit only by ambient or indirect lighting. 33
4.8 The metallic roughness material model calculates the final output for the specular component by mixing dielectric and metallic values. The image on the left shows the model rendered as purely dielectric, while the image on the right shows the model as purely metallic. 34
4.9 Images of each of the three lights lighting the model individually. The model rendered in these images is lit by the full Cook-Torrance BRDF including all three components: ambient, diffuse and specular. 34
4.10 Complete render of the `DamagedHelmet` model using the implemented SVMV application, containing all of the aforementioned components. 35

Chapter 1

Introduction

This thesis concerns the visualization, or rendering, of 3D models, specifically their material properties. One of the primary goals of computer graphics is rendering naturalistic scenes lit by light that behaves in a realistic way. Over the years great progress has been made regarding this goal, including the development of ”Physically Based Rendering,” further referred to as PBR.

PBR is a somewhat loose term that refers to a collection of rendering techniques meant to represent and render lifelike interactions between object surfaces and light. It allows artists to set properties regarding various qualities of a model's surface that are contained in a material, which is consequently applied to the modeled object. These properties are then processed and used as parameters in the renderer's shaders, in order to accurately calculate the final appearance of the rendered object. As in the name, in order to be considered ”physically based”, the rendering techniques must adhere to real physical theories of the behavior of electromagnetic radiation.

The standard of computer rendered visuals has risen with time and the increasing performance and availability of computer hardware. New technologies have also had an impact, making physically based techniques more viable. Modern graphics API's, such as Vulkan, model formats that contain descriptions of material properties and advanced software for artists has made PBR the baseline for contemporary realistic graphics.

The goal of this thesis is to research PBR techniques and develop an application that can load, process, and render 3D models in the glTF format, utilizing the Vulkan graphics API. The application is a model viewer, allowing users to load their glTF models from storage and see a visualization of their model, including its material properties, using PBR in real time.

Chapter 2

Theory

This chapter contains first an overview of the physical behavior of light and its interactions with matter, along with an explanation of the necessary definitions and concepts related to it. The following section concerns the development of physical models used in computer graphics, specifically ones relevant for physically based rendering. The Cook-Torrance model and its basis in microfacet theory [3] is presented with examples for each of its component functions. Next is a history of PBR with a focus on real-time rendering applications and the various models and algorithms used, followed by a section on indirect lighting. The two final sections concern the glTF model format and the Vulkan API respectively.

2.1 Physics of light propagation

All visual perception is caused by light illuminating objects and then being absorbed by the human eye. The word ”light” refers to a specific subset of electromagnetic radiation know as the visible spectrum. Electromagnetic radiation is a type of radiation which permeates throughout the universe via subatomic particles known as photons, whose creation is primarily caused by shifts in electron energy levels. Photons exhibit wave-particle duality meaning each photon is characterized by a wave-length or frequency property. The visible spectrum covers wave-lengths roughly between 360nm and 800nm, but may vary person to person since vision as a sense is tied to each person's individual perception. When photons interact with matter, they are either absorbed and manifest as heat, or are released and continue to travel through space depending on the energy state of the matter's electrons and the frequency associated with the photon. Each material absorbs photons of specific wave-lengths, which allows the brain to differentiate them by assigning a spectrum of colors to a spectrum of wave-lengths (the aforementioned visible spectrum). For example, a purely green object would absorb all photons not exhibiting the frequency perceived as green, while ”green photons” would be emitted away from the object causing it to take on that color. Color is not the only property the human brain uses to facilitate visual perception however, the density of photons emitted from some light source is also a visual factor and is perceived as brightness. The visual appearance of an object depends therefore on the frequencies and number of photons extant from the object's surface and the material properties of the object. [9] [11]

Light is always traveling through some medium, whether that be air, water or even a solid object. When it reaches the boundary between two media, a portion of it is reflected

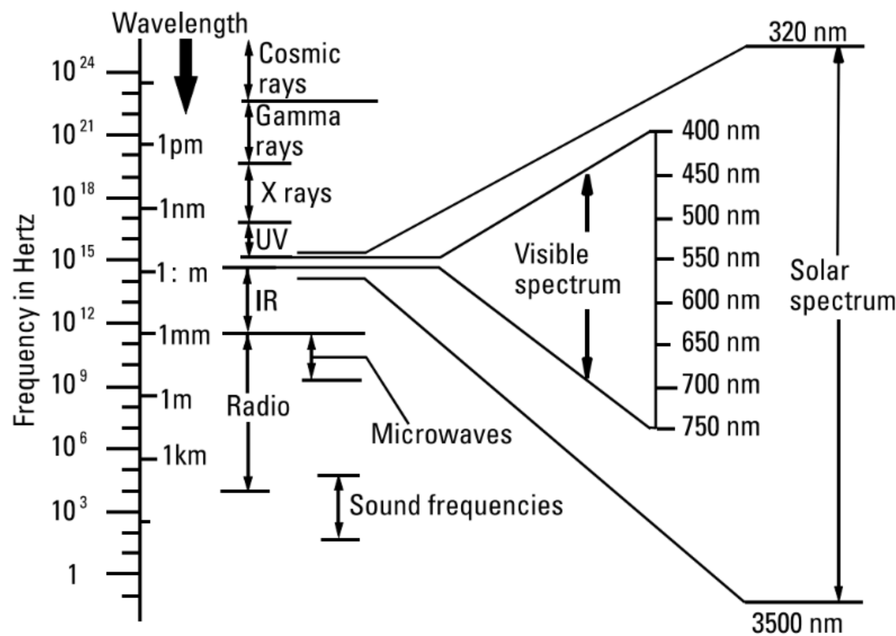


Figure 2.1: Breakdown of frequencies exhibited by electro-magnetic radiation, with the visible spectrum highlighted as a subset of the „solar spectrum“. [9]

away from the boundary, while the rest is either absorbed or enters the second medium, changing its direction according to Snell's law [11]. The ratio of reflected to transmitted light can be calculated using Fresnel's equations [11] and is dependent on the refractive indices of the media and the angle between the incident light ray and the normal vector of the boundary. The amount of reflected light proportional to transmitted light increases as the vector of incidence approaches the grazing angle. This behavior of light traveling through and between media is referred to as light propagation.

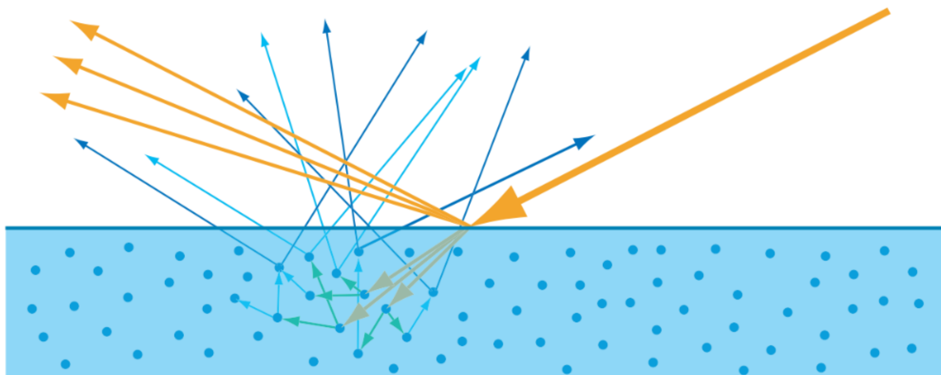


Figure 2.2: As light impacts the boundary of two media, a portion of it is reflected approximately following the angle of reflection, while another portion enters the medium, bouncing inside of it between atoms before exiting the surface. [1]

As mentioned, the creation and absorption of photons relates to the excitement of electrons bound to atoms in affected media, and various types of matter result in varying

behavior of excited electrons, causing physical materials to be divided into three categories [11]:

- **Dielectrics** covers electric insulators such as glass, water or paper. These materials tend to have a low index of refraction and thus much of the incident light tends to be transmitted into dielectric media, depending on the angle of incidence.
- **Conductors** such as silver or gold reflect almost all incident light as electrons are able to move larger distances, causing the energy of transmitted photons to decay and manifest as heat. Photons are thus unable to penetrate surfaces of conductors to any great depth.
- **Semiconductors** can have properties of both dielectrics and conductors.

When discussing the behavior of light for the development of physical models, certain conventions regarding the directionality of light rays are used. In these models there are three important vectors all originating from the point on the surface who's interactions with light are of concern, namely the vector towards the light source, the vector towards the eye or camera and the halfway vector between the two. The first of the three appears in literature as either L for light or ω_i (i as in „incident"). Despite the fact that the photons are emitted from the light source and impact the surface, conventionally all of these vectors' origins are at the point on the surface and are normalized. The second vector, towards the eye or camera is marked V for viewer or ω_o (o as in „outgoing"). The halfway vector is marked either H of ω_h and can be calculated using the following equation [3]:

$$H = \frac{V + L}{|V + L|} \tag{2.1}$$

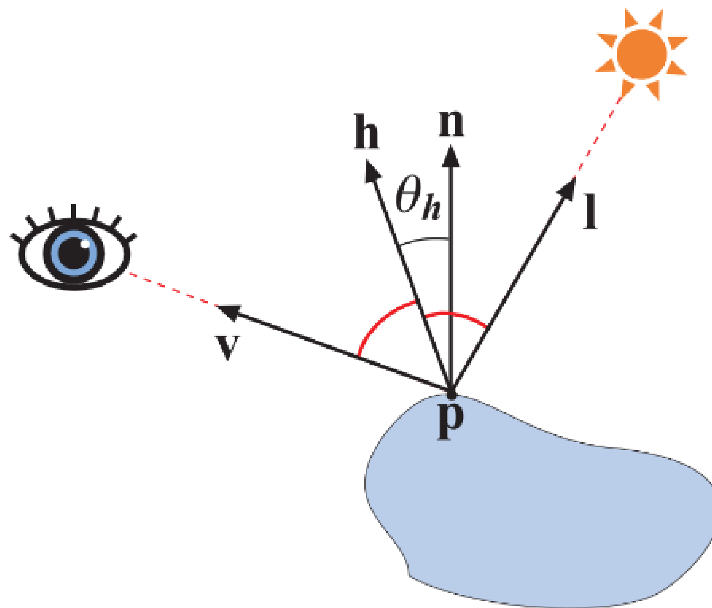


Figure 2.3: The light (l), view (v), normal (n) and halfway vector (h) shown in relation to the point (p) on a two dimensional sufrace. [1]

Studying the physical behavior of electro-magnetic radiation is the concern of **radiometry**, which has been employed in computer graphics research and is particularly relevant when defining physically based models used for realistic visualization. When describing the physics and mathematics behind rendering 3D scenes, four physical quantities of radiometry and the concept of a solid angle are used [9]:

- **Solid angle** (ω, in sr) is the extension of the planar angle into three dimensions. The planar angle subtends the arc on a unit circle formed by a two dimensional object projected onto the circle, measured in radians. The solid angle similarly subtends the projection of a three dimensional object onto a unit sphere and is measured in steradians.
- **Radiant flux** (Φ, in $watts$), also referred to as radiant power, defines the amount of energy emanating from a light source per unit time and is used to define the strength and color of light sources when rendering (defined as an RGB triplet for example).
- **Radiant intensity** (I, in $watts/sr$) is only relevant for point light sources and specifies the energy emanating from the light source in a particular direction. The direction is specified by an area on the surface of a unit sphere centered on the light source.
- **Irradiance** (E, in $watts/m^2$) measures the density of radiant flux incident on or passing through a surface.
- **Radiance** (L, in $watts/(sr * m^2)$) measures radiant flux emanating from a point p in a specific direction ω, it is the most relevant quantity of radiometry, as radiant flux and irradiance can be calculated from radiance via integration over a surface or solid angle.

Photometric equivalents to these quantities are luminous flux, luminous intensity, luminance and illuminance. [9] [11]

2.2 Models of light propagation for use in rendering

The visualization of 3D scenes requires not only the presence of a 3D model, composed of geometric data forming its mesh and material properties, but also sources of light. During early 3D rendering efforts various algorithms were devised to visualize rendered objects and various methods of implementing light sources were employed. In 1986 James T. Kajiya generalized these efforts into what is now commonly known as the „rendering equation.“ It has since served as the mathematical basis of all rendering models and takes this form [7] [11]:

$$L_o(p, \omega_o) = \int_\Omega f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) n \cdot \omega_i d\omega_i \tag{2.2}$$

The outgoing radiance from some point p in the direction ω_o is equal to the sum of all irradiance from all light sources in all incoming directions ω_i normalized by the cosine term $n * \omega_i$ and multiplied by the bidirectional reflectance function $f_r(p, \omega_i, \omega_o)$, commonly abbreviated as BRDF. The cosine term accounts for Lambert's cosine law, where the irradiance incident on a surface is scaled by the cosine of the angle between ω_i and the normal vector of the surface, due to the fact that when the surface covered by incident light is projected

onto the unit sphere surrounding the light source, its projection covers a greater surface which subsequently lowers the irradiance incident on it [9]. Most important for material visualization is the BRDF term, which describes the statistical distribution of reflected light incident at some point p along the vector ω_i, meaning it describes the physical interactions between the surface and incident light. For each input outgoing vector ω_o it returns the probability that light is reflected in that direction. Many different BRDFs exist, employed in various cases depending on the required visual fidelity, available material properties, computational limitations, aesthetic considerations and other factors. [11]

When attempting to visualize various material surfaces accurately, a BRDF must be chosen that adheres to and is based on real physical properties of light propagation. Real physical behavior of light is too complex to simulate accurately for rendering purposes, and while models that take into consideration the wave properties of light have been proposed, models more appropriate for real-time rendering are generally based on geometric optics [17] [3]. Geometric optics represents light as rays, ignoring light's behavior as a wave, which simplifies the calculations and has not been found to have a significant impact on the visual quality of the rendered image [8].

Several models have been proposed for real-time rendering purposes, the first relevant model focused on achieving more realistic visuals being Phong shading, which represents extant light as a combination of three components: diffuse (representing transmitted light), specular (representing reflected light) and ambient (representing directionless indirect lighting as a combination of both diffuse and specular). In the Phong shading model the final output is a linear combination of a perfect diffuse model and a perfect specular model, with ambient lighting added as a constant factor to ensure surfaces that are not directly lit are visible [1].

Figure 2.4: The Phong shading model is a combination of an ambient, diffuse and specular term, each representing different interactions between light and the rendered object. [1]

This linear mix of a diffuse and specular term, with the ratio either set by a designer or calculated using the Fresnel equations is used by all the models mentioned in this text.

$$f_r(\omega_i, \omega_o) = k_d f_d(\omega_i) + k_s f_s(\omega_i, \omega_o) \tag{2.3}$$

The factors k_d and k_s represent proportions of diffuse and specular light respectively, where $k_d = 1 - k_s$. As seen in equation 2.3 above, the diffuse and specular terms are calculated independently, meaning they are each calculated using a specific BRDF and defined by specific models. As mentioned above, k_s, being the ratio of reflected light, can be calculated using the Fresnel equations with the indices of refraction of the two media.

[1]Taken from the Basic Lighting chapter at learnopengl.com

A very close approximation of the Fresnel equations discovered by Schlick is often used in computer science, referred to as „Schlick's approximation" [13]:

$$F(\omega_i, \omega_h) = f_0 + (1 - f_0)(1 - |\omega_i \cdot \omega_h|)^5 \tag{2.4}$$

The resulting term $F(\omega_i, \omega_h)$ is equivalent to the term k_s from equation 2.3. The two parameters of the function are the direction of incident light ω_i and the halfway vector ω_h. The factor f_0 is the measured or designed ratio of reflected to transmitted light of the surface at normal incidence, or when $\omega_i = n$.

In the metallic-roughness material model, the fresnel functions for dielectric and metallic materials are considered separate and are used to calculate output values for each material type. The metallic factor is then used to mix these results accordingly. This is due to the fact that dielectric materials tend to have indices of refraction (IORs) around 1.5, while the IOR's of conductors vary. For example, according to the glTF specification [5], the fresnel function for dielectrics must use a fixed f_0 value of 0.04, while conductor fresnel function uses the color value, or albedo, of the material. [5]

Material	f_0 **(Linear)**	f_0 **(sRGB)**	**Color**
Water	0.02, 0.02, 0.02	0.15, 0.15, 0.15	
Plastic / Glass (Low)	0.03, 0.03, 0.03	0.21, 0.21, 0.21	
Plastic High	0.05, 0.05, 0.05	0.24, 0.24, 0.24	
Glass (High) / Ruby	0.08, 0.08, 0.08	0.31, 0.31, 0.31	
Diamond	0.17, 0.17, 0.17	0.45, 0.45, 0.45	
Gold	1.00, 0.71, 0.29	1.00, 0.86, 0.57	
Silver	0.95, 0.93, 0.88	0.98, 0.97, 0.95	
Copper	0.95, 0.64, 0.54	0.98, 0.82, 0.76	
Iron	0.56, 0.57, 0.58	0.77, 0.78, 0.78	
Aluminium	0.91, 0.92, 0.92	0.96, 0.96, 0.97	

Table 2.1: Table containing R_F (0°) values for various materials. The top five material are dielectrics, and their R_F values are generally low, around 0.04. The bottom five materials are conductors exhibiting much higher and more varied R_F values. [1]

The diffuse component represents transmitted light, where photons enter the medium and bounce between atoms inside of it, eventually exiting it at potentially any point on the surface of the object and in any possible direction. A common model for calculating extant diffuse light is the Lambertian diffuse model. A Lambertian surface is perfectly matte, causing all incident light to enter the object without any reflection. The light bounces inside the medium to the point where extant photons lose all directionality, meaning extant photons are distributed equally among all possible directions. The lambertian diffuse BRDF is very simple [11] [2]:

$$f_d(\omega_i) = \frac{c}{\pi} \tag{2.5}$$

The cosine term is omitted due to its presence in the rendering equation. The c term is the albedo term which essentially describes the color of the extant photons from the surface, generally in the form of an RGB triplet sampled from a texture for example, while π is present in the denominator to normalize the output. This model yields good results for light near normal incidence, however it has been found not to adhere to experimental data

when the light direction is near grazing angle. This is somewhat resolved when using it in combination with a specular model where the ratio is calculated using Fresnel's equations [2]. Other diffuse models exist, for example the Oren-Nayar model [10] which takes into account retro-reflection relevant when visualizing rough surfaces or Burley's diffuse model, but they have not been found to raise visual quality to a significant enough degree to warrant their usage [14] [8]. Another phenomenon not represented by the Lambertian model is subsurface scattering where light passes through certain materials, visible for example when shining a light through skin or wax. Implementing subsurface scattering is a complex and difficult task and requires a great amount of computing power as photons traveling through objects of varying sizes must be simulated. It does however result in more realistic visuals in regards to certain materials, especially human skin. [11]

While the simple Lambertian model is sufficient for representing transmitted light, the specular component is more complex. In the model proposed by Phong, what is referred to as off-specular tint is simulated via a cosine function raised to some power. This is somewhat realistic for specific materials but generally does not adhere to experimental data, causing all surfaces to take on a plastic like appearance. A more realistic model based on research from optics was proposed by Cook and Torrance for use in computer graphics, namely the microfacet model. The specular component for realistic surfaces cannot be modeled as a perfectly smooth reflector, since in practice real physical surfaces are never perfectly flat, so the surface is represented as a collection of microscopic mirror-like surfaces named microfacets. [17]

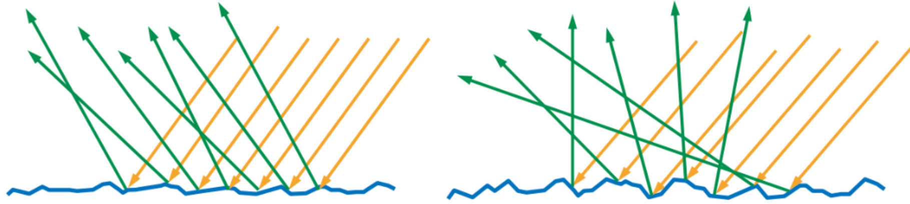


Figure 2.5: A visualization of the surface of an object at a microscopic scale. The microsurface illustrated on the image on the left is less rough than the microsurface on the image on the right. Reflected light rays are scattered in various directions depending on the roughness of the surface. [1]

Each microfacet is oriented in a certain direction and is generally identified using its normal vector. The statistical distribution of microfacet orientations depends on the nature of the physical surface's real shape (in the PBR workflow this is represented as one floating point value referred to as „roughness") and is given by the **normal distribution function** D, also referred to as an „NDF." [3] [17]

When a ray of light is reflected off of a flat surface, the angle between the incoming ray vector and the normal vector is equal to the angle between the outgoing ray vector and the normal. In order to absorb the ray of light, the vector from the point on the surface to the camera sensor must be equal to the outgoing light vector. This means that when given a light vector, a view vector and a point on a surface, it is possible to find a surface orientation that would reflect the incoming light directly towards the camera using the **halfway vector** mentioned in section 2.1. The halfway vector is halfway between the viewing vector and the light vector, calculated by adding the two vectors together and normalizing the output. It is then possible to use the desired halfway vector, or desired microsurface normal to calculate

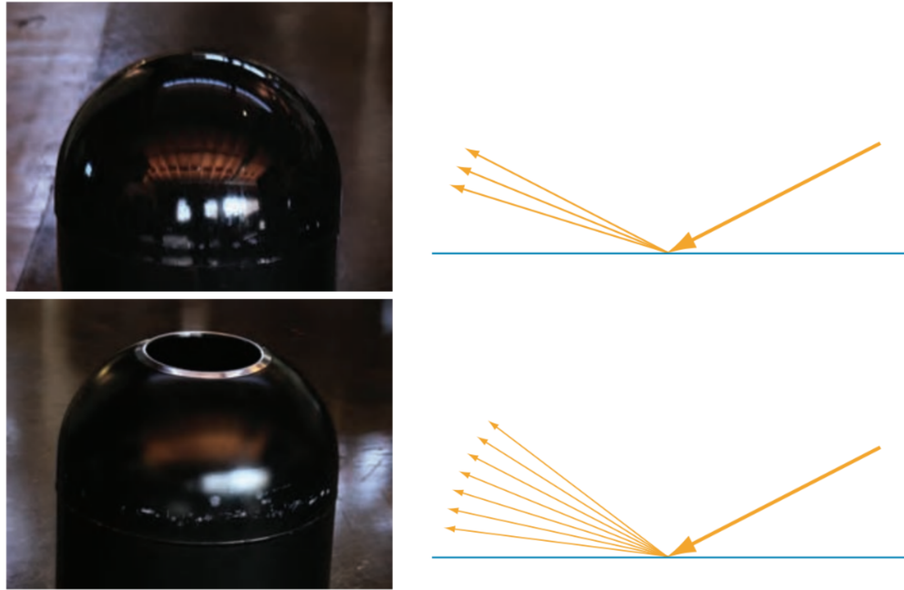


Figure 2.6: The top left image shows a round object with very low micro surface roughness, with incident light scattering visualized by the image on the top left. The bottom two images show the same phenomenon except with a higher micro surface roughness. [1]

the proportion of the surface's microfacets oriented in such a way that the incoming light ray is reflected directly towards the eye or camera. The normal vector of the entire surface is referred to as the „macrosurface normal.“ The center of the specular highlight on a surface will be very bright due to the fact that it is statistically likely for the microfacets in that area to be oriented similarly to the macrosurface normal of the surface, while the off-specular tint is caused by a smaller probability that the microsurface normal will be oriented not to the macrosurface normal but to the desired halfway vector. The shape and size of the specular highlight is thus dependent on the normal distribution function.

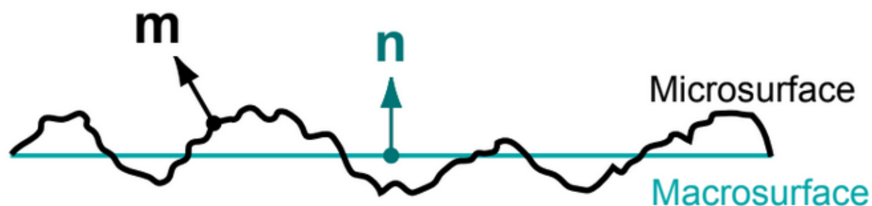


Figure 2.7: This image illustrates the difference between the microsurface in black and the flat macrosurface in blue. Two distinct normal vectors, one for the microsurface (m) and one for the macrosurface (n) are shown originating in points on their respecting geometric surfaces. [17]

While various NDFs exist, one commonly seen in popular graphics applications (as shown in section 2.3) is GGX, developed by Walter et al. in 2007 [17]. Their findings were independently discovered and presented already in 1975 by Trowbridge and Reitz [16], when they published their own paper. For this reason, the resulting normal distribution function is referred to as either Trowbridge-Reitz, GGX or Trowbridge-Reitz/GGX. In comparison

to other NDFs like the one presented by Beckmann, the specular highlights of GGX feature longer tails meaning they adhere more closely to measured data [17]. Its mathematical form is:

$$D(\omega_h) = \frac{\alpha^2 \chi^+(\omega_h \cdot n)}{\pi((\omega_h \cdot n)^2(\alpha^2 - 1) + 1)^2} \tag{2.6}$$

The α parameter specifies the roughness of the surface, while χ^+ represents the positive characteristic function, which returns either 1 when the input is positive or 0 if it is negative or zero. Developers generally map some external parameter, $\alpha = roughness^2$ in the glTF specification for example [5], to the α parameter in the function depending on their development workflow and aesthetic considerations.

There are two scenarios where this model fails: either the incident ray of light never reaches the microfacet's surface due to being blocked by another microfacet, meaning it is never reflected towards the camera despite the target microfacet being oriented appropriately (referred to as shadowing). Another possibility is that a ray of light never reaches its target, the camera, also due to being blocked by another microfacet after reflection (referred to as masking).

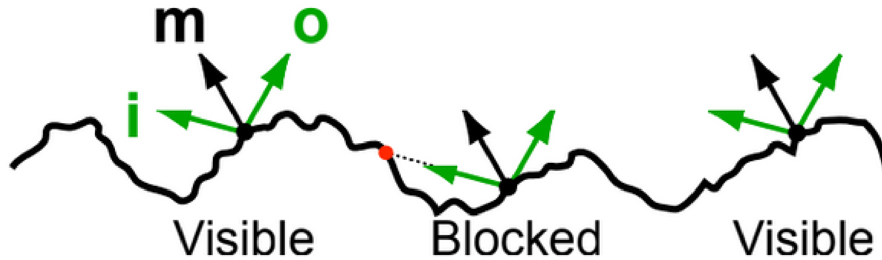


Figure 2.8: A visualization of the phenomenon of shadowing, where the reflected vector in the middle never reaches the camera due to being blocked by the microsurface. The vector m represents the microsurface normal, while the vectors i and o represent incoming and outgoing light respectively. [17]

A separate function, namely the **geometric function**, is needed to account for the phenomena. Shadowing and masking are seen mostly at grazing angle and the closer the incident ray angle gets to it, the more prevalent it is. A common method of acquiring shadowing functions comes from Smith, where the function is separated into equal shadowing and masking functions [15] [6]:

$$G(\omega_i, \omega_o) = G_1(\omega_i)G_1(\omega_o) \tag{2.7}$$

The left term accounts for shadowing, having as a parameter incident light ω_i, while the right term accounts for masking with the parameter being the direction to the camera ω_o. G_1 can be calculated using the chosen normal distribution function using this formula [17]:

$$G_1(\omega) = \chi^+\left(\frac{\omega \cdot \omega_h}{\omega \cdot n}\right)\frac{1}{1 + \Lambda(\omega)} \tag{2.8}$$

The positive characteristic function χ^+ present in the formula is the sidedness agreement and ensures rays do not pass through the surface itself [17]. The $\Lambda(\omega)$ term is dependent on the NDF. Applying it to the Trowbridge-Reitz/GGX microfacet distribution results in:

$$G(\omega_o, \omega_i) = \frac{2\ |n \cdot \omega_i|\ \chi^+(\omega_h \cdot \omega_i)}{|n \cdot \omega_i| + \sqrt{\alpha^2 + (1-\alpha^2)(n \cdot \omega_i)^2}} \frac{2\ |n \cdot \omega_o|\ \chi^+(\omega_h \cdot \omega_o)}{|n \cdot \omega_o| + \sqrt{\alpha^2 + (1-\alpha^2)(n \cdot \omega_o)^2}} \tag{2.9}$$

Once a model of light reflection has been chosen, along with the appropriate normal distribution function and geometric function, the specular BRDF can be assembled. Specular BRDFs based on microfacet theory generally take on the following appearance [17]:

$$f_s(\omega_i, \omega_o) = \frac{D(\omega_h)G(\omega_i, \omega_o)}{4(\omega_i \cdot n)(\omega_o \cdot n)} \tag{2.10}$$

The numerator contains the chosen normal distribution function and its related shadowing function, whose output is normalized by the terms in the denominator. The specular term is much more complex to compute than the diffuse term and requires not only the light vector as a parameter but also the view vector. The fresnel function $F(\omega_i, \omega_h)$ is oftentimes included in the specular BRDF but since it is already included in equation 2.3 as the factor k_s it is not included here.

An overview of the history of the development of various BRDF models is presented in Appendix A of Burley's article on Physically Based Shading at Disney [2].

2.3 Physically based rendering

The term PBR encompasses multiple techniques within the field of computer graphics, not only related to rendering but also asset creation. In 2012, Disney introduced their modernized production workflow titled physically based shading, intended to allow for more realistic visuals in their animated films. The introduced workflow utilized many of the techniques mentioned in the previous sections alongside older ones that had been in use in real-time computer graphics already, such as normal mapping.

Figure 2.9: This still from the Disney produced film Wreck-It Ralph, showcasing the results of their new physically based shading workflow. [2]

Disney's new workflow served as the basis for Epic Games' Unreal Engine 4's PBR renderer, similarly intended to allow for more realistic visuals in video games. Since then the collection of techniques has been generalized and adopted widely in computer graphics. In order to qualify as „physically based“, an application's renderer must adhere to the following:

- **Utilize a microfacet model** as explained in section 2.2. The calculation of the specular term must involve the Cook-Torrance based combination of a normal distribution function and a geometry function in order to realistically visualize the specular highlight.
- **Be energy conserving**, meaning no more energy is emitted from the rendered surface than is incident on it.
- **Use a physically based BRDF** originating from a physically based model such as geometric optics.

One of the earliest video games to utilize a PBR workflow was Ryse: Son of Rome, initially exclusive to the Xbox One platform from Microsoft, developed by Crytek using CryEngine 3. Its main goal was to present the graphical capability of the next generation of consoles through high visual fidelity and realistic material visualization. Earlier games were forced to set unrealistic parameters for their materials when trying to approach a realistic look, setting extremely high IOR values for wooden materials for example. Crytek's goal was cleaner looking, less stylized visuals where the materials could be clearly seen. To accomplish this they adopted a physically based shading model using the aforementioned Cook-Torrance model and the GGX normal distribution function for reasons presented in section 2.2. As the fresnel term they used Schlick's approximation and for the geometric function they utilized a simplified version of the separable Smith function, namely the Schlick-Smith approximation. An approximation of the Oren-Nayar diffuse model was chosen to serve as their diffuse BRDF, despite it only being marginally different from the Lambertian model, in order to more realistically visualize rough materials common in their game such as concrete or rough stone. [14]

Another important step in the adoption of PBR for real-time graphics was its inclusion in Unreal Engine 4, presented at the SIGGRAPH conference in 2013. The developers of Unreal Engine, Epic Games, wished to update their material model to align with rising industry standards and allow artists to improve their workflow. Inspired by Disney's paper on physically based shading, Epic Games decided to adopt PBR for their engine. The shading model was similar to Crytek's, except for their diffuse model being Lambertian, having tested other models and deeming them not worth the cost, and minor adjustments to the fresnel term and the geometric function. [8]

The game engine Unity would include a physically based shading model in 2014 as part of Unity 5, replacing their older Blinn-Phong shading model. Also inspired by Disney's workflow, they would adopt the Cook-Torrance microfacet model similar to those mentioned above. One major difference is they opted to utilize the Blinn-Phong NDF instead of GGX. Today, Unity 6 uses GGX for its shading model in its „Universal Render Pipeline,“ while its shading model in the „Built-In Render Pipeline“ is comprised of the GGX NDF, the Smith-GGX geometry function and the Fresnel-Schlick approximation. [12]

A current example, the Godot game engine, uses PBR for its default material model in its latest version. In Godot 4.4 the default option for diffuse shading is Burley's model from

Disney, however they do provide Lambertian as a second option, while the specular model is referred to as „SchlickGGX,“ which they describe as the most common model used by 3D PBR engines. [2]

2.4 Models of material properties

In order to render realistic looking 3D models, the models used must contain properties describing the physical surfaces that comprise them. The nature of the properties depends on the model format, rendering application, and requirements regarding the realism of the rendered image. These properties include number values and textures, such as ”normal maps” or ”occlusion maps” and serve as parameters for the application to render the models as their designer intended. Real-time 3D rendering only became capable of visualizing realistic looking models, more specifically utilizing PBR for material visualization, in the late 2000’s with the onset of more powerful and more programmable GPU’s in combination with new rendering techniques that allowed for quicker lighting calculations by making use of approximation. In the 2010’s the PBR workflow as outlined by Disney became common in the industry, and served as the basis for Unreal Engine 4’s lighting system. [2] [1] [8]

In practice, the model’s geometry and material are either separate, created and loaded independently and assembled together using the rendering application, or included together in one singular file, available when using formats including FBX, glTF and USD for example. Each format contains its own model for specifying materials, with varying amounts of represented detail. The material model contained in glTF files is explicitly designed to facilitate PBR (specifically the metallic-roughness material model) and in its specification contains outlines regarding its intended use and visual appearance for the purposes of consistent visualization across various applications. It represents a common general material model used for realistic rendering and is comprised of the following [5]:

- **Base color factor and texture**, also referred to as ”albedo” or ”diffuse,” defines the color of the surface, meaning how the material affects the spectral distribution of incident light at normal incidence f_0. When defined as a factor (a quartet of floating point numbers defining the red, green and blue components of an RGB triplet and an alpha value for transparency), it defines the appearance of the entire surface as one color.

- **Normal texture** is a two dimensional array of three dimensional vectors specifying the normal vector of the surface at that point. Each point on the surface is represented by a pixel in the texture with the red, green and blue channels defining the X, Y and Z components of the normal vector. The vectors are defined in what is know as tangent space with the origin being the $[0, 0]$ coordinate of the texture and the z basis vector pointing away from the surface, meaning they must first be transformed using the geometric normal vectors and tangent vectors contained in the polygon’s vertex attributes.

- **Occlusion texture** specifies which areas of the model are occluded by the geometry of the modeled object, corners or indents for example. These areas receive less indirect light and appear darker. Only one value is necessary for each pixel, which is used as a factor when applying indirect lighting. This means the occlusion texture is often

[2] https://docs.godotengine.org/en/stable/tutorials/3d/standard_material_3d.html#shading

Figure 2.10: An example of the glTF material model. The left side shows the geometry of the model, while the right side shows slices of the various textures available as material properties. [5]

combined with the roughness and metallic textures in one image referred to as an "ORM" texture.

- **Emissive factor and texture** is used when the surface or parts of the surface emit light. Can be implemented as an area light or simply added to the final color of the rendered surface.
- **Metallic factor and texture** define the metallic factor of the surface. In the metallic workflow meant to be used with glTF models, both dielectric and metallic BRDF's are calculated and their values mixed according to the metallic factor of the surface. Only one value is required, meaning only one RGB channel is used when in the form of a texture.
- **Roughness factor and texture** contain values defining the roughness of the surface material used when calculating the specular component of reflected light. Just like occlusion and the metallic factor, this property only requires one value and therefore takes up only one channel of the texture.

2.5 glTF

The GL Transmission Format, or glTF is a free specification for the storage of 3D models focused on minimizing the size of the stored assets and simplifying the process of exporting and transferring assets between applications. Developed by the Khronos group, the glTF model format was first presented under the name WebGL Transmission Format in 2012 and released in 2015 as the glTF 1.0 specification.

The main focus of glTF is runtime efficiency and the reduction of processing overhead when handling 3D models in graphical applications. The goal is to create a format suitable for a wide range of software, including web applications and software on mobile. A glTF file is comprised of a JSON formatted file (`.gltf`) defining all of the structures forming the model, such as nodes, meshes, material and others, binary files containing the actual data of the model (`.bin`) and image files containing the textures used in the model. Models can also come in `.glb` files, where all of the data defining the model in the aforementioned files is combined into a singular file. A glTF file can contain information regarding the structure of the scene, geometry data, data related to skeletal animation, material properties, textures, cameras and animation. The feature set of glTF can be extended via various optional extensions.

The scene, or model, as it is stored in the glTF file is at the lowest level comprised of structures referred to as „primitives". Each primitive is designed to require one graphics API draw call to render, consisting of vertex attributes, specifically position, normal, tangent attibutes and one or more sets of texcoord, color, joints and weights attributes respectively. The final two relate to skeletal animation. Other properties defined in the primitive structure are the indices of the geometry, the material of the primitive and the mode which defines the primitive's topology type. Primitives are defined inside a mesh structure along with the name of the mesh. Meshes are defined in a separate section and one mesh can be referenced by index in a node along with a camera, information related to skeletal animation, the node's transform information, the node's name and the node's child nodes. Nodes are organized into a node tree defined by the scene structure, with each scene containing a name and one or more root nodes.

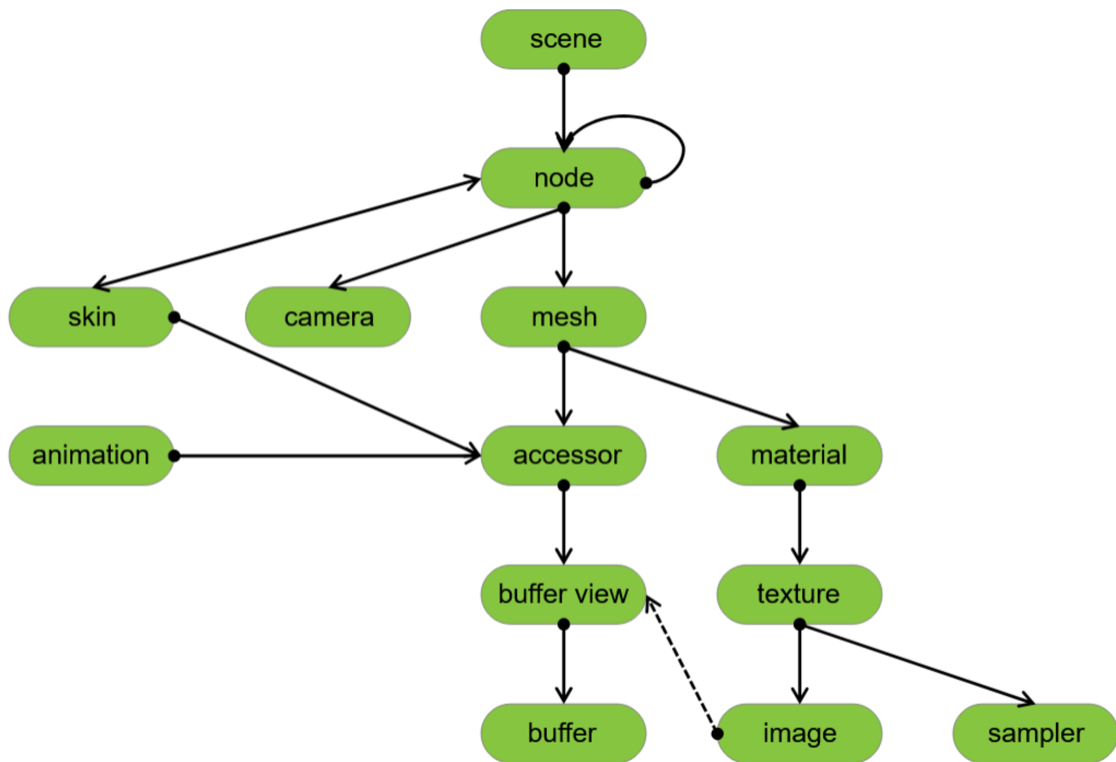


Figure 2.11: glTF model format structure. [5]

Materials, similarly to meshes and nodes, are defined in their own section of the glTF file and consist of a name, information regarding the normal, occlusion and emissive texture and the emissive factor. Also present are fields related to transparency such as alpha mode, alpha cutoff and a boolean specifying whether the primitive should be rendered double sided or not. Most relevant for this thesis is the presence of the `pbrMetallicRoughness` property. Contained within are the base color texture and factor, roughness metallic texture and individual roughness and metallic factors. The function of these properties was explained in section 2.2.

All data in a glTF model is contained in structures referred to as „buffers," which are defined as blobs of binary data. Various kinds of data may be contained in a buffer in varying formats. In order to access the data in a buffer another structure must be used. namely the „buffer view". Buffer views specify an offset into the buffer, defining where the relevant data begins, a byte length specifying the length of the chunk of relevant data and a byte stride field. To efficiently process and load model data from the buffers, yet another structure is defined in glTF. When a vertex attribute is specified in a primitive for example, it references an accessor structure which then references a specific buffer view, along with an offset into the buffer view, the type of the attribute (`VEC3`, `SCALAR`, etc.) and the type of the elements in the buffer (`float`, `unsigned int`, etc.). Accessor also include the count and maximum and minimum values relevant for the accessed data. [5]

2.6 Vulkan

The origins of the Vulkan API start in 2014, when the Khronos group called for the development of a next-generation graphics API. Vulkan would answer the demand for a low level API without the built-in overhead featured in OpenGL or Direct3D 11 and allow developers more room for performance improvements as the standard for computer graphics in video games rose and developers were beginning to reach the limits of the available options. AMD had also started developing an exclusive API with the game company DICE named Mantle, meant to offer the same benefits as Vulkan later would, while Microsoft had started working on their equivalent, Direct3D 12. Vulkan, however, would be open-source and portable, meant to be adopted widely for various platforms. The main distinguishing features of modern graphics APIs are explicit control over resources, highly programmable graphics pipelines and explicit synchronization between the CPU and the GPU. [4]

Chapter 3

Design

The goals of this thesis is the implementation of the aforementioned PBR algorithms and the development of a demonstrative application meant to show the results. The demo application, named „SVMV“ for Simple Vulkan Material Visualization, is a 3D model viewer able to load, process and render models in the glTF format. Rendering is done using the Vulkan API and the application is designed and implemented using modern programming techniques. A graphical user interface provides information about the rendered model and allows the user to select a file using a file explorer menu.

3.1 Application design

A model viewer must be able to accomplish multiple varying tasks using a variety of third-party libraries, it must be efficient enough to render models and their material properties using physically based rendering techniques and due to the complexity of such an application should be developed using modern technologies and methods. In order to handle all of this, the application should be divided into independent components responsible for the individual tasks of the application.

- **glTF Importer & Loader**, necessary for loading the model's data into memory and converting it from the structures of the glTF format to structures defined by the application for use by the renderer. The importer allows access to the data in the glTF file, while the loader's role in the conversion between structures as the data in the file is organized to take up the least amount of space and is unfit for use in a rendering application.
- **Vulkan Renderer**, being the most important part of the application, initializes the graphics API (Vulkan in this case), defines the rendering pipeline, loads all the necessary resources and submits commands for the GPU to execute.
- **Scene Graph**, forming a graph of the internal model related structures. Organized into a graph of nodes with each containing meshes and vectors for accessing all the meshes, materials and potentially other scene objects such as lights.
- **Application Manager**, a class representing the entire application and all of its components. The application manager is responsible for the initialization, coordination and termination of all the individual components such as the loader, renderer, etc.

- **Input Manager** responsible for the handling of keyboard and mouse inputs related to the window of the application.
- **Window Manager** initializing and managing the window of the application. Requests the creation of a window from the operating system and manages all the signals and requests related to it.
- **GUI Manager**, for graphical user interface, displays the interface to the user and allows them to interact with the application via interactive UI elements.

3.2 Loading models

The application must be able to import and then load the data of glTF models from storage. The model format includes geometry, material properties, but also animations and data related to skeletal animations. Not all of the data potentially included in the glTF file is relevant for the application. Custom internal structures must be developed to store only data relevant for the purposes of the application. These structures are organized into a graph during loading, and so the internal structure of the loaded model must be translated to the internal scene graph.

3.3 Shading model

The shading model chosen for the application must take into account primarily the material model of the glTF format. According to the glTF specification, the material properties defined in a glTF model are intended to be used with a metallic-roughness PBR model. The Cook-Torrance model based on microfacet theory was presented in section 2.2 with examples for each of its mathematical components. In section 2.3, the history of PBR in real-time rendering, specifically video games, was presented with multiple examples and an overview of the shading algorithms used in the industry up to today. Taking all this information into consideration, the chosen shading model is comprised of the following:

- **Lambertian diffuse model**, chosen for its simplicity and acceptable results.
- **Trowbridge-Reitz/GGX normal distribution function**, used in each of the three major publicly available game engines, namely Unreal Engine, Unity and Godot. It provides closer results to measured data than the alternatives, such as Beckmann, and a geometric function can be derived for it using Smith's method.
- **Smith-GGX geometric function**, derived from the GGX NDF, is commonly paired with it.
- **Fresnel-Schlick approximation**, used by all the graphics applications mentioned in this text,

This represents a commonly seen shading model featured in graphics applications such as popular publicly available game engines as mentioned in section 2.3 and adheres to the glTF specification.

Chapter 4

Implementation

This chapter contains an overview of the technology used to develop the application, followed by a detailed explanation of each of its components. At the end, a demonstration of the application rendering a sample model is presented.

4.1 Used technology

The application is written in C++20, targeting the Microsoft Windows operating system version 11. Building and compiling the project requires the use of the CMake[1] build system with the provided `CMakeLists.txt` file included in the root directory of the application's code. All source code files are included in the `src` directory, resources including shader code and sample models are in the `res` directory and a portion of the libraries used to build the application are included in the `thirdparty` directory.

The application was developed with the intent to utilize modern programming and software development techniques and principles such as RAII[2], including the use of Vulkan classes from the `vk::raii` namespace. A portion of the components forming the application (see diagram in figure 4.1) were implemented from scratch, namely the application class itself, the scene graph and its associated structures, the loader, the vulkan renderer and the input handler. Various external libraries were also used:

- **GLFW**[3] is an open source C library intended to simplify development by streamlining the process of window creation and management. It also allows for the use of callback functions to handle events related to the window and the creation of the window surface necessary for Vulkan.

- **glm**[4] is a header-only C++ library containing structures and functions related to linear algebra for use in graphical applications.

- **tiny-gltf**[5] is a header-only C++ library used for loading glTF models into memory and allowing the application to access their data.

[1]https://cmake.org/
[2]https://en.cppreference.com/w/cpp/language/raii
[3]https://www.glfw.org/
[4]https://github.com/g-truc/glm
[5]https://github.com/syoyo/tinygltf

- **Vulkan**[6], or the Vulkan API, is a modern graphics and compute API used to program GPUs. It is a modern graphics API, alongside Direct3D 12, meaning it allows for greater control over GPU resources at the cost of complexity.
- **shaderc**[7] is a collection of tools related to the compilation of shader code. It includes `glslc`, a command line application used to compile GLSL or HLSL shader code to SPIR-V and a C library that allows for the use of `glslc` during run-time. Developed by Google, it is also available as part of the Vulkan SDK.
- **Vulkan Memory Allocator**[8] is a popular library designed to streamline operations related to memory management when developing applications using the Vulkan API. Used by the likes of Blender, Godot Engine and popular video games such as Baldur's Gate III. Commonly abbreviated as VMA, it is also included in the Vulkan SDK.
- **vk-bootstrap**[9] is a library used to streamline and greatly simplify operations related to the initialization and management of necessary Vulkan API structures, intended to avoid a great amount of boilerplate code.
- **Dear ImGui**[10] is a graphical user interface C++ library, primarily suited for use in game engines and other real-time graphical applications. Its goal is to be as simple and quick to integrate into the application as possible.
- **ImGuiFileDialog**[11] is a publicaly available implementation of a file selection dialog window using ImGui.
- **MikkTSpace**[12] is a C library used to generate tangent vertex attributes for models that utilize normal maps. Written by Morten S. Mikkelsen, it is now free to use for the public and is the recommended process for tangent generation according to the glTF 2.0 specification [5].

4.2 Scene graph

All the relevant data loaded from a glTF model file is stored in structures defined by the application. The primary structure representing the entire loaded scene is name `Scene` and contains the root node of the loaded scene alongside two vector containing all the `Mesh` and `Material` structures of the scene. `Mesh` objects contain only a vector containing all of the meshes primitives. Each `Primitive` then contains its indices, a vector of `Attribute` objects and the primitive's material. Vertex attributes are stored in the aforementioned `Attribute` structures, which contain the attribute data buffer, the type of attribute and other fields related to the data buffer.

`Material` objects contain two strings, one for the name of the material type (specifying the pipeline intended to render the material, „glTFPBR" for example) and the other for the specific material name. A vector of `Property` objects is also present, where the `Property`

[6] https://www.vulkan.org/
[7] https://github.com/google/shaderc
[8] https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator
[9] https://github.com/charles-lunarg/vk-bootstrap
[10] https://github.com/ocornut/imgui
[11] https://github.com/aiekick/ImGuiFileDialog
[12] http://www.mikktspace.com/

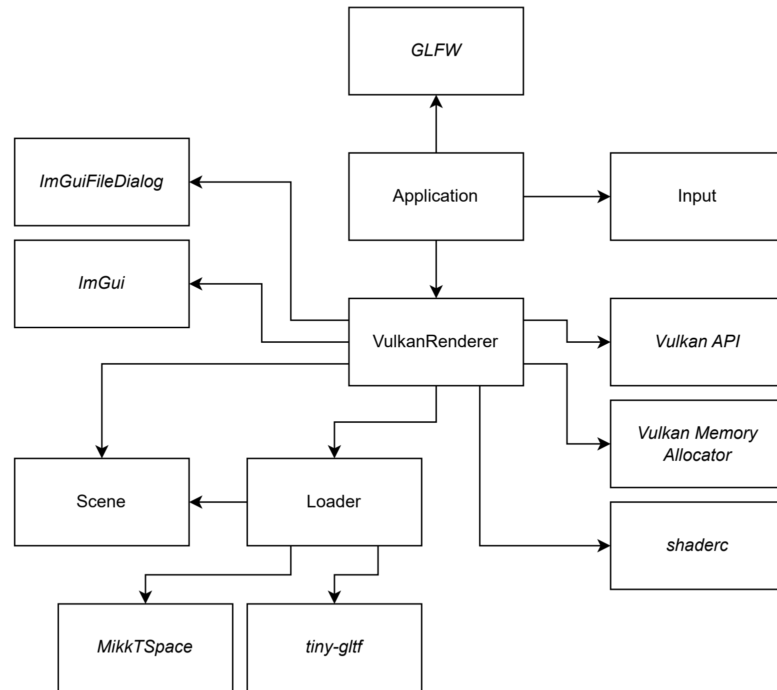


Figure 4.1: Diagram showing the relationships between individual components and libraries forming the application. External libraries are designated by an italicized name.

structure contains specific material properties such as the base color texture, emissive factor and others.

The scene is organized into a directed acyclic graph, with the graph's nodes represented by the `Node` structure. Each `Node` contains a transform matrix, a vector of child nodes and the mesh.

4.3 Loader

The loader is used via the loadScene function defined in the `Loader` namespace. Unlike the other components, the loader is not defined as a class, due to the fact that no internal state needs to be kept between individual loading operations.

The user first specifies the glTF file to be loaded using its file path as a string. The string is provided to the loader which first imports the model using the **`tiny-gltf`** library as its importer. The importer loads the file into memory and provides structures with which to access the data. As the model file contains various properties related to the geometry of the file, its material properties, animations, data related to skeletal animation and cameras, the loader processes only the properties relevant for the application. All data related to the geometry of the meshes contained in the model file must be loaded and comes in the form of vertex attributes. The attributes loaded are position, normal, tangent, texcoord_0 and color_0. A loaded model must contain positions and normals in order to be considered valid by the application. Tangent attributes are often omitted from model files due to the fact that they are generated using recognized algorithms on import. For this reason, if a loaded

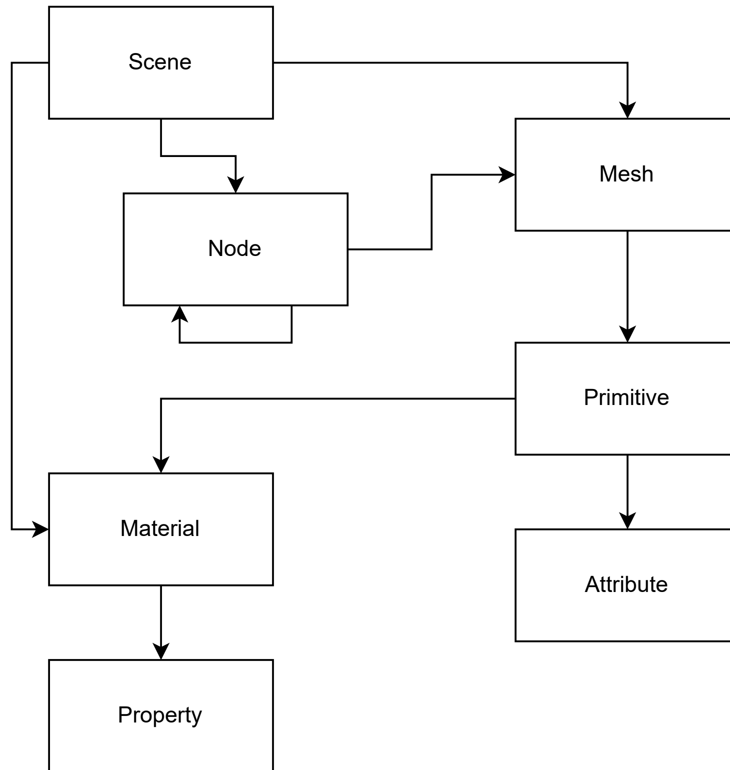


Figure 4.2: Diagram of the structures defined and used by the application to store the loaded scene, or model, in memory.

model is missing tangent attributes, they are generated using the **MikkTSpace** algorithm and library. Missing texture coordinate and color attributes are considered valid and their absence is handled by the renderer. All material properties are loaded, including all the textures defined in the specification, and a Material object is created and defined with the material name „glTFPBR" and the loaded properties. Missing textures are replaced with default textures by the renderer, as mentioned below. The various „factor" properties are loaded from the model file or set to default values.

For performance reasons, the loader first processes all the meshes in the imported model, followed by the materials. These are then stored as vector fields containing pointers to the loaded data structures in the scene graph object. The imported file's node hierarchy is then processed and converted, ensuring the internal structure of the model and all the node transforms remain as intended, and the loaded meshes and materials are connected to their appropriate nodes. Once the process is complete, the final result is a scene graph object containing all the relevant data in structures defined for the purposes of the application.

4.4 Transferring scene data to the GPU

Before the individual fragment colors can be calculated, the rendered scene's data must first be loaded into GPU memory in an appropriate format using memory related structures

defined by the Vulkan API. A traditional option is to use a vertex buffer with a predefined vertex attribute format where attribute data relevant for each vertex is provided for use in the vertex shader. This limits accessible vertex attribute data to the current vertex being processed and to the defined format. For example, if the format is defined as a vector of three floats representing the position attribute followed by the same type vector for the color attribute, models lacking vertex color data would be required to fill the appropriate memory with a default value for each vertex, or a separate graphics pipeline for models containing position attributes only would be necessary. In practice, models can contain vertex attribute data for position, normal, tangent, color and texture coordinates. The adopted solution involves the use of shader storage buffer objects (SSBO for short), which are GPU memory buffer accessible in shaders via a memory address allowing for access of individual elements using pointer arithmetic with the built-in `gl_VertexIndex` variable.

The renderer is responsible for the transfer of model data to GPU memory and accomplishes this task in three phases. First the scene is preprocessed, where all vertex attributes and meshes in the loaded model are counted and GPU memory buffers are allocated for individual attributes, model transform matrices and normal matrices. The necessary buffers, including their appropriate staging buffers (used for efficient transfer of data from the CPU to the GPU), are then organized into a vector of `VulkanAttribute` objects in a `VulkanScene` object, which represents the loaded scene in GPU memory. The following phase involves the creation of `VulkanDrawable` objects, each requiring one draw command call to render. The actual data of the vertex attributes is pushed onto the relevant staging buffers and the memory addresses in the attribute buffers are stored for each drawable. The normal matrix is calculated here and likewise pushed a into staging buffer alongside the model matrix. A descriptor set containing the material properties of each mesh is created and stored in the `VulkanDrawable` object as well. The final phase involves the transfer of data from each staging buffer into the GPU-only buffers.

Loading the material properties is the responsibility of the `VulkanGLTFPBRMaterial` class, alongside the creation of the Vulkan graphics pipeline used for rendering glTF models and material data for each processed material. Loaded material properties are used to generate a descriptor set for each processed material, which is then bound for each rendered primitive. The descriptor set layout contains six bindings, the first being a uniform buffer used to store all non-image properties, such as the base color factor, metallic factor, roughness factor etc. The next five bindings are all used to store combined image samplers for the base color texture, normal texture, metallic-roughness texture, occlusion texture and the emissive texture. Supplied material objects are processed property by property, starting with the non-image properties. These are stored in a uniform buffer in VRAM in the `MaterialUniformParameters` structure. Following this a VulkanImage object, responsible for the creation and storage of image data on the GPU, is created for each of the image properties of the material. If any property is undefined, the `VulkanGLTFPBRMaterial` class provides placeholders. Each of these is then written to the generated descriptor set in their appropriate bindings.

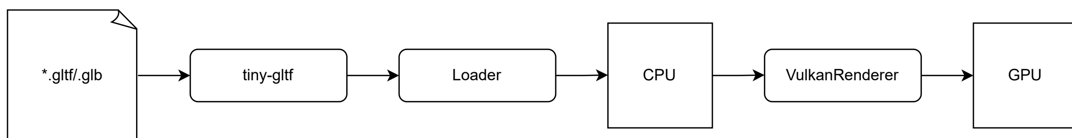


Figure 4.3: Diagram showing the full process of loading a glTF model file.

4.5 Rendering process

Once scene data is loaded into VRAM, the renderer can begin issuing commands to the GPU. In order to avoid having the CPU wait for one frame to finish presentation before a new frame can be rendered, the renderer in the implemented application manages multiple frames „in-flight“. Each iteration of the rendering loop refers to a particular frame using its index, which is cycled through at the end of each iteration. First, the CPU waits for the GPU to finish rendering the current framebuffer via a fence. Once the active framebuffer has finished rendering the CPU must again wait for an image from the swapchain to become ready to be written to. Next the descriptor sets containing information about the camera and lights in the scene are updated, after which the commands to be submitted to the graphics queue can be recorded.

The first command begins the render pass by specifying the framebuffer to be written to, the render area and the clear color. Next the renderer iterates through every material context in the loaded scene (e.g. „glTF PBR“) and binds the appropriate graphics pipeline, index buffer and descriptor sets containing information related to the camera and the lights in the scene. The renderer then iterates over every `VulkanDrawable` object in the material context, binding its descriptor set containing its material properties before filling the `PushConstants` structure with addresses for the drawable's vertex attributes and matrices in their respective shader storage buffers to be accessed in the vertex shader. These are then pushed to the GPU along with the actual draw call. The render pass is then ended, followed by the recording of commands related to the rendering of the **ImGui** user interface.

The recorded commands are then submitted to the graphics queue for execution alongside two semaphores, one ensuring the GPU waits until the swapchain image is ready for writing and another to be signaled when the framebuffer rendering is finished. The renderer then submits a request to the present queue specifying the swapchain, an index for the swapchain's image and the semaphore signaled when framebuffer rendering is finished. If during this process the swapchain becomes invalid (the window is resized for example), the swapchain is recreated with the appropriate parameters. Finally, the frame index is incremented.

4.6 Lights

Considering the application is intended for the purposes of rendering individual models in order to visualize the model's material model, the following light system was designed. The renderer contains a `VulkanLight` object, responsible for managing all the light sources in the scene. Three orbiting point lights are always present, each lighting the model with its own flux and from its own position. Indirect light is handled via the presence of a field containing the color of indirect lighting, which is used as the ambient factor to calculate each fragment's final output color.

All light-related data is stored in a uniform buffer at binding 0 of descriptor set 1, which is updated to reflect requested changes at the beginning of each rendering loop. The position and flux of the point lights and the ambient light can be changed using the graphical user interface and updates in real time.

4.7 Implemented shading model

As mentioned in chapter 3.3, the implemented shading algorithms are based on the Cook-Torrance microfacet model in combination with a Lambertian diffuse model. The specific algorithms used are the GGX NDF alongside a Smith-GGX joint masking-shadowing function and the Fresnel-Schlick approximation.

Before each fragment's color can be calculated in the fragment shader, the vertex shader must first process each vertex and supply necessary data to the fragment shader. The vertex shader code for the PBR pipeline is contained in the file `gltf_pbr_vert.glsl`. After the loaded scene's data is transferred to the GPU (as explained in section 4.4), the data is provided to the vertex shader using a variety of methods. Information related to the camera and lights is provided separately via uniform buffers at binding 0 in descriptor set 0 and descriptor set 1 respectively. Vertex attribute and matrix shader storage buffers are accessed via buffer references provided using push constants. The vertex shader first transforms the position of the vertex to viewport coordinates and sets the built-in `gl_Position` variable, after which the other four attributes are handled. An output variable is defined for the geometric normal in world space, texture coordinates, color attribute, position in world space and the position of the camera in world space which are then used as parameters for the fragment shader. For the color attribute a conditional statement is used to check whether the buffer address is valid, meaning it is not 0, and can be used to access attribute data. For texture coordinates and color, the vertex shader simply loads the attribute from the buffer. The geometric normal is transformed to world space using the normal matrix and and is used along with the tangent attribute to generate the tangent-bitangent-normal matrix (also referred to as the TBN matrix), which is necessary for the use of normal vectors sampled from a normal map. Bitangents are calculated using normals and tangents according to the glTF specification [5]. The renderer uses a technique where instead of transforming the sampled normal vector into world space in the fragment shader, meaning an operation involving matrix multiplication must be calculated for every fragment, the position of the camera and light source is transformed to tangent space in the vertex shader and all light calculations in the fragment shader are conducted in tangent space.

The fragment shader first samples the normal map as specified in the glTF specification [5] in order to acquire the macrosurface normal of the rendered surface.

$$\omega_n = texture(normal_map, uv) * 2.0 - 1.0 \tag{4.1}$$

Next, the metallic-roughness, occlusion and emissive textures are sampled, after which the flux of each light source, the light's radiance value and the fragment's distance to it are calculated. The three vectors mentioned in 2.1 are calculated next, namely ω_i, ω_o and the halfway vector ω_h, all defined in tangent space. The first component of the BRDF that is calculated is a simple ambient light factor scaled by the occlusion factor sampled from the occlusion texture. It is then multiplied by the base color of the surface and added to the final output color value. Second is the diffuse component, calculated using the Lambertian diffuse BRDF (see equation 2.5):

```
vec3 brdf_d_lambert(in vec3 base_col) {
    return base_col / PI;
}
```

Listing 4.1: Lambertian diffuse BRDF.

Being the most complex of the three, the specular BRDF is calculated last. The Cook-Torrance specular BRDF is calculated using the function `brdf_s_cook_torrance`, which accepts as parameters all the aforementioned vectors and the α value, which in the glTF material model is defined as $\alpha = roughness^2$.

```
vec3 brdf_s_cook_torrance(in vec3 L, in vec3 V, in vec3 N, in vec3 H, in float alpha) {
    return vec3(1.0) * ((d_ggx(N, H, alpha)
    * g_smith_t_r_ms(N, V, L, H, alpha))
    / (4.0 * max(dot(N, L), 0.0) * max(dot(N, V), 0.0) + 0.0001));
}
```

Listing 4.2: Cook-Torrance specular BRDF.

The function is equivalent to the general Cook-Torrance BRDF as seen in equation 2.10, with the normal distribution function `d_ggx` calculated first.

```
float d_ggx(in vec3 N, in vec3 H, in float alpha) {
    float alpha_sq = alpha * alpha;
    float dot_N_H = max(dot(N, H), 0.0);
    float dot_N_H_sq = dot_N_H * dot_N_H;

    float numerator = alpha_sq;
    float denominator = PI * (dot_N_H_sq * (alpha_sq - 1.0) + 1.0) * (dot_N_H_sq * (
    alpha_sq - 1.0) + 1.0);

    return numerator / denominator;
}
```

Listing 4.3: GGX normal distribution function.

The implemented function is based on the GGX NDF seen in equation 2.6. The numerator and denominator of the equation are calculated separately and their division is returned. The result is then multiplied by the geometric function `g_smith_ggx_ms`.

```
float g_smith_ggx_ms(in vec3 N, in vec3 V, in vec3 L, in vec3 H, in float alpha) {
    return g_smith_ggx(V, N, H, alpha) * g_smith_gxx(L, N, H, alpha);
}
```

Listing 4.4: Smith-GGX masking-shadowing geometry function.

This represents a multiplication of the separable components of the Smith-GGX joint masking-shadowing function and returns the multiplication of the two components according to equation 2.7. The individual geometric functions are calculated using the `g_smith_ggx` function.

```
float g_smith_ggx(in vec3 A, in vec3 N, in vec3 H, in float alpha) {
    float alpha_sq = alpha * alpha;
    float dot_N_A = max(dot(N, A), 0.0);

    float numerator = 2.0 * dot_N_A;
    float denominator = dot_N_A + sqrt(alpha_sq + (1.0 - alpha_sq) * dot_N_A * dot_N_A);

    return numerator / denominator;
}
```

Listing 4.5: Smith-GGX G_1 function.

Similarly to the NDF, the numerator and denominator are calculated individually. The values calculated using the above functions are then mixed according to the specified metallic-roughness shading model intended to visualize glTF materials.

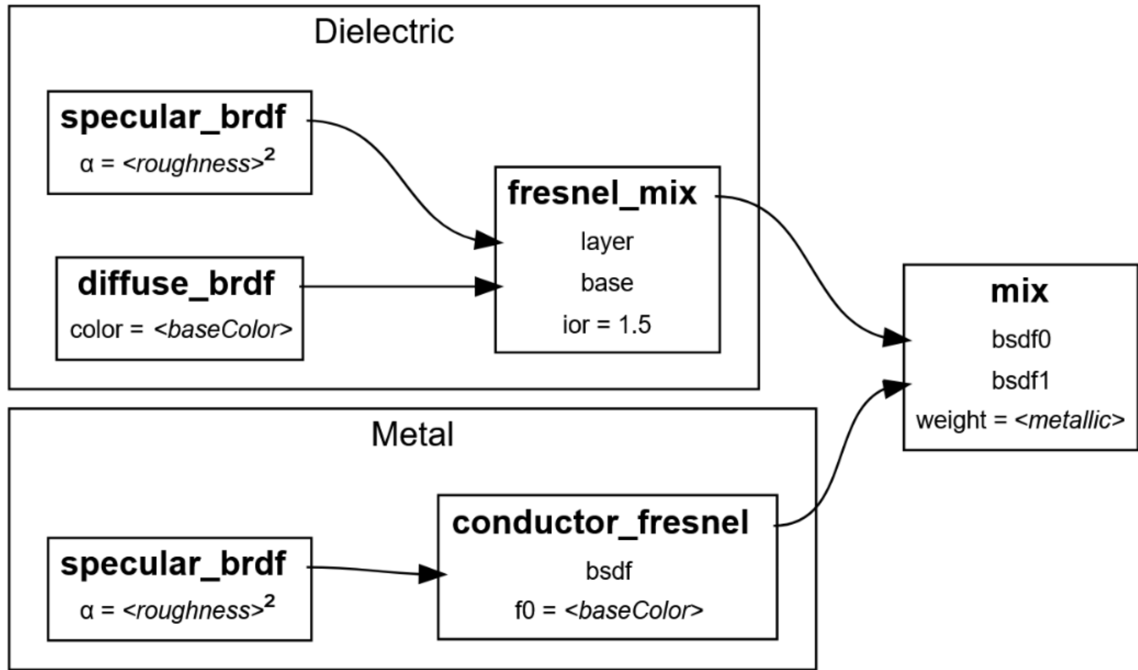


Figure 4.4: Diagram showing a high-level view of the intended glTF metallic-roughness shading model. [5]

As seen in figure 4.4, separate Fresnel functions are needed for conductors and dielectric materials. The difference is minor, with the dielectric Fresnel function using a fixed IOR value of 1.5 and the metallic function instead using the base color of the surface. The implemented functions are `f_schlick_metal` and `f_schlick_dielectric` respectively.

```
vec3 f_schlick_metal(in vec3 H, in vec3 V, in vec3 f_0) {
    return f_0 + (1.0 - f_0) * pow(1.0 - max(dot(V, H), 0.0), 5);
}

float f_schlick_dielectric(in vec3 H, in vec3 V, in float f_0) {
    return f_0 + (1.0 - f_0) * pow(1.0 - max(dot(V, H), 0.0), 5);
}
```

Listing 4.6: Fresnel-Schlick approximation for conductors and dielectrics.

Once all of the above values have been calculated, they are mixed according to the specified shading model.

```
vec3 ambient_factor = light_params_buf.ambient.rgb * light_params_buf.ambient.w * (
    occlusion_factor);

vec3 diffuse = brdf_d_lambert(base_col);
vec3 specular_0 = brdf_s_cook_torrance(ts_L_0, ts_V, ts_N, ts_H_0, roughness * roughness);

float f_s_dielectric_0 = f_schlick_dielectric(ts_H_0, ts_V, 0.04);
float f_d_dielectric_0 = 1 - f_s_dielectric_0;
vec3 f_s_metallic_0 = f_schlick_metal(ts_H_0, ts_V, base_col);

vec3 col_dielectric_0 = f_d_dielectric_0 * diffuse + f_s_dielectric_0 * specular_0;
vec3 col_metallic_0 = f_s_metallic_0 * specular_0;

```

```
13 vec3 final_col_0 = mix(col_dielectric_0, col_metallic_0, metallicity);
```

Listing 4.7: Calculation of the final result of the BRDF.

Each variable with a light index at the end of its name, for example `specular_0` is calculated separately for each of the three point lights, due to the changing incoming light vector ω_i (`ts_L_0` in the shader code for example). The calculated final color for each light source is then added to the final output color along with the color sampled from the emissive texture, the ambient lighting factor and mixed with the color from the color vertex attribute.

```
1     out_col = vec4(ambient_factor * base_col, 1.0);
2     out_col = out_col + vec4(final_col_0 * radiance_0 * max(dot(ts_N, ts_L_0), 0.0), 0.0);
3     out_col = out_col + vec4(final_col_1 * radiance_1 * max(dot(ts_N, ts_L_1), 0.0), 0.0);
4     out_col = out_col + vec4(final_col_2 * radiance_2 * max(dot(ts_N, ts_L_2), 0.0), 0.0);
5     out_col = max(out_col, vec4(emissive_col, 1.0));
6     out_col = out_col * col_0;
```

Listing 4.8: Calculation of the final output color.

4.8 Results

This section contains images meant to isolate and visualize individual components forming the final visual output. It also serves as a visual demonstration of the rendering techniques mentioned in previous chapters that were implemented for the application. The model chosen for the demonstration was acquired from the set of sample glTF models available for free by the Khronos group.[13] The specific model is named „Damaged Helmet,“ [14] created by `theblueturtle_` under the Creative Commons Attribution - Non Commercial license. This model is lit by three orbiting lights, with their light colors being magenta, orange and light green respectively. The color of the ambient light is a pale blue with a low ambient strength. In terms of material properties, the model contains five texture properties, the base color (referred to as „albedo“) texture, the metallic and roughness texture, ambient occlusion texture, normal texture and emissive texture (see section 2.4 for an overview).

Figure 4.5: Images of the rendered model with the output color set to values sampled from texture material properties. From left to right these are the normal, roughness and metallic texture. The vectors sampled from the normal texture are in tangent space.

[13] glTF sample models available at https://github.com/KhronosGroup/glTF-Sample-Models
[14] Damaged Helmet by `theblueturtle_`

As seen in figure 4.5, the first texture properties sampled in the fragment shader are the normal map (if the model contains one) and the texture containing metallic values in the blue channel and roughness values in the green channel. If the material does not contain a normal map, a field in the descriptor set regarding material properties is set to inform the shader to instead use the geometric normal. The geometric normal is transformed to tangent space in the vertex shader and passed to the fragment shader in case of an absence of the normal map.

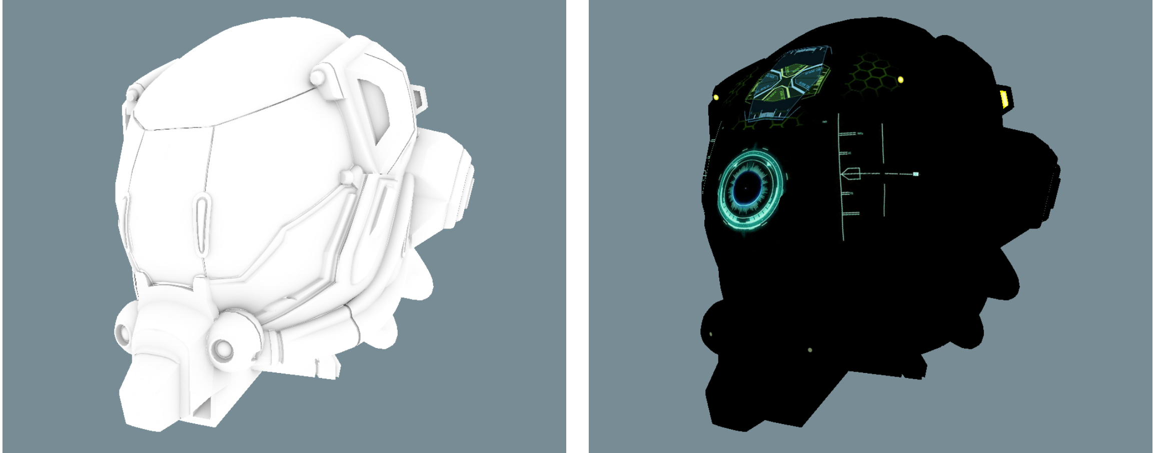


Figure 4.6: Images of the rendered model with the output color set to values sampled from the two remaining texture properties. The are, from left to right, the ambient occlusion texture and the emissive texture.

The remaining two sampled textures are the ambient occlusion texture and emissive texture as seen in figure 4.6. The ambient occlusion texture is sampled and used to adjust the indirect lighting (or ambient lighting) factor in order to simulate darker corners. The emissive texture is added to the final output color after all other lighting is calculated via the `max()` function and is meant to represent the model's light emitting surfaces.

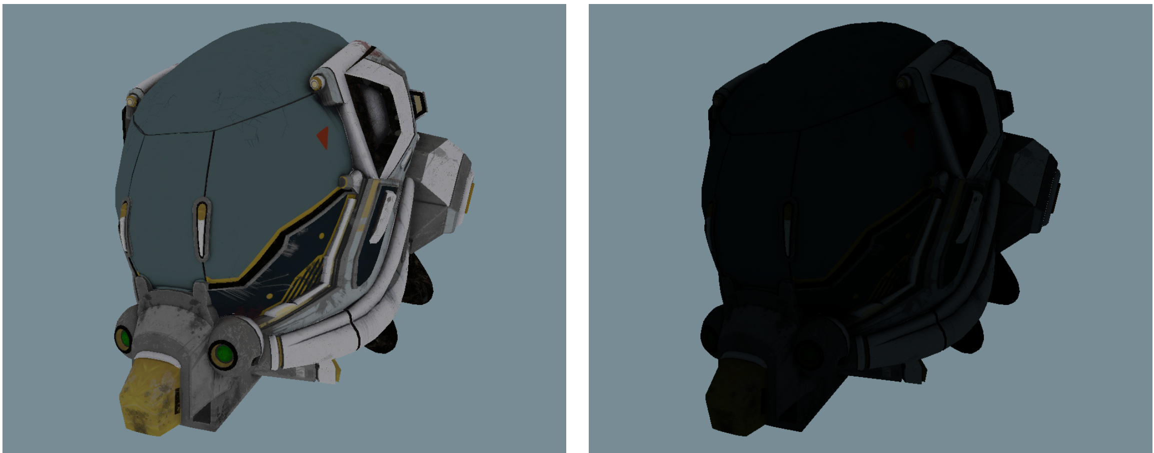


Figure 4.7: The left image shows the model rendered using only the Lambertian BRDF without accounting for the cosine factor (mentioned in section 2.2), while the right image shows the model lit only by ambient or indirect lighting.

Once the ambient and diffuse components of the shading model are calculated (as seen in figure 4.7), they can be combined with the specular component. The remaining specular component is the most complex and must be evaluated separately for each light.

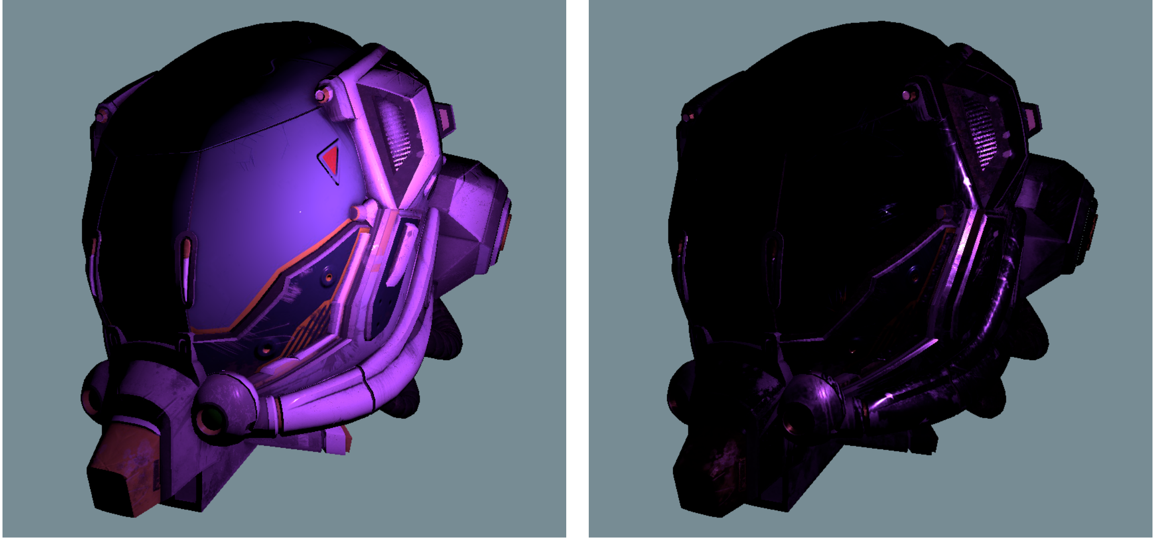


Figure 4.8: The metallic roughness material model calculates the final output for the specular component by mixing dielectric and metallic values. The image on the left shows the model rendered as purely dielectric, while the image on the right shows the model as purely metallic.

As seen in figure 4.8, the resulting specular component for each light is a mix of the results of the dielectric and metallic specular BRDFs. The ratio for the mix is given by the metallic value sampled from the metallic roughness texture (seen in figure 4.5).

Figure 4.9: Images of each of the three lights lighting the model individually. The model rendered in these images is lit by the full Cook-Torrance BRDF including all three components: ambient, diffuse and specular.

All of these are then combined together to form the final output color of each fragment as seen in figure 4.10. The first two lights, magenta and orange respectively can be seen on the left side of the model, with specular highlights appearing on various surfaces with varying material properties. The model is lit by the third, light green colored light from the front. Grime and surface imperfections caused by using the normal texture are seen

all over the model, while differences in roughness and the result of mixing dielectric and metallic surface properties are seen in the varying forms of specular highlights. The surface of the visor of the helmet prominently displays the emissive texture,

Figure 4.10: Complete render of the `DamagedHelmet` model using the implemented SVMV application, containing all of the aforementioned components.

Chapter 5

Conclusion

The goal of this thesis was the study and implementation of shading models used for material visualization. Before any work on the implementation could be done, a great amount of time was spent researching the many topics relevant for the thesis. A study of the underlying physical principles of light propagation, followed by a history of lighting models for the purposes of computer graphics was conducted first, followed by research regarding the use of these models in real-time applications. The Vulkan API and glTF model format were studied as well, with the intricacies of Vulkan causing work on the implementation to start later than desired. Once the research phase was over, work on the application could begin in the form of experimentation and planning, after which the general design of the application could form.

In the end, the goal of the thesis was completed, with the final application able to render glTF models containing the relevant material properties. The user is able to move the camera, adjust all the light properties in the scene in real time and load models using a graphical user interface.

In terms of extending the application, I would consider two options: implementing image based lighting, which would improve the visual results especially when it comes to smooth metallic surfaces and implementing other shading models, such as a simple blinn-phong model for non-PBR models and other artistic shading models.

Bibliography

[1] Akenine Möller, T.; Haines, E. and Hoffman, N. *Real-Time Rendering.* 3rd ed. A K Peters/CRC Press, 2008. ISBN 978-1568814247. Available at: http://www.realtimerendering.com/.

[2] Burley, B. Physically Based Shading at Disney. *ACM SIGGRAPH Courses*, 2012. Available at: https://disney-animation.s3.amazonaws.com/library/s2012_pbs_disney_brdf_notes_v2.pdf. Practical Physically Based Shading in Film and Game Production.

[3] Cook, R. L. and Torrance, K. E. A Reflectance Model for Computer Graphics. In: *ACM SIGGRAPH Computer Graphics.* 1982, vol. 16, no. 3, p. 307–316.

[4] Ghavam, K. *The Vulkan SDK – Where We Started – Where We Are Going* Presentation at SIGGRAPH 2024, Vulkan BOF: Vulkan, Forging Ahead. July 2024. Available at: https://www.lunarg.com/see-lunarg-at-the-vulkan-forging-ahead-session-at-the-khronos-bof-during-siggraph-2024/. Presented on July 31, 2024, at the Hyatt Regency Denver, Capitol Ballroom 1-3.

[5] Khronos Group. *GlTF 2.0 Specification.* Khronos Group, 2017. Available at: https://www.khronos.org/registry/glTF/specs/2.0/glTF-2.0.html. Version 2.0, released June 5, 2017.

[6] Heitz, E. Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs. *Journal of Computer Graphics Techniques (JCGT)*, 2014, vol. 3, no. 2, p. 32–91. Available at: http://jcgt.org/published/0003/02/03/.

[7] Kajiya, J. T. The Rendering Equation. *ACM SIGGRAPH Computer Graphics*, 1986, vol. 20, no. 4, p. 143–150.

[8] Karis, B. Real Shading in Unreal Engine 4. In: *ACM SIGGRAPH Courses.* 2013. Available at: https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf. SIGGRAPH 2013 Course: Physically Based Shading in Theory and Practice.

[9] McCluney, W. R. *Introduction to Radiometry and Photometry.* 2nd ed. Artech House, 1994. ISBN 978-0890066782.

[10] Oren, M. and Nayar, S. K. Generalization of Lambert's Reflectance Model. In: *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH).* 1994, p. 239–246.

[11] Pharr, M.; Jakob, W. and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. 3rd ed. Morgan Kaufmann, 2023. ISBN 978-0128006450. Available at: https://www.pbr-book.org/.

[12] Pranckevičius, A. Physically Based Shading in Unity. In: *Game Developers Conference (GDC)*. 2015. Available at: https://www.gdcvault.com/play/1022106/Physically-Based-Shading-in.

[13] Schlick, C. An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum*, 1994, vol. 13, no. 3, p. 233–246.

[14] Schulz, N. Moving to the Next Generation: The Rendering Technology of Ryse. In: *ACM SIGGRAPH Talks*. 2014. Available at: https://advances.realtimerendering.com/s2014/index.html. Presented at SIGGRAPH 2014.

[15] Smith, B. G. Geometrical shadowing of a random rough surface. *IEEE Transactions on Antennas and Propagation*, 1967, vol. 15, no. 5, p. 668–671.

[16] Trowbridge, T. S. and Reitz, K. P. Average irregularity representation of a rough surface for ray reflection. *Journal of the Optical Society of America*, 1975, vol. 65, no. 5, p. 531–536.

[17] Walter, B.; Marschner, S. R.; Li, H. and Torrance, K. E. Microfacet Models for Refraction through Rough Surfaces. *Rendering Techniques (Proceedings of EGSR)*, 2007, p. 195–206.