NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Physically Based Lighting Models for Real Time Graphics Engines

Konstantinos A. Kyriakos

Supervisor: **Theoharis Theoharis,** Professor N.K.U.A.

ATHENS

JULY 2022

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μοντέλα Φωτισμού Βασισμένα στη Φυσική για Μηχανές Γραφικών Πραγματικού Χρόνου

Κωνσταντίνος Α. Κυριακός

Επιβλέπων: **Θεοχάρης Θεοχάρης,** Καθηγητής Ε.Κ.Π.Α.

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2022

BSc THESIS

Physically Based Lighting Models for Real Time Graphics Engines

Konstantinos A. Kyriakos
S.N.: 1115201600079

Supervisor: **Theoharis Theoharis,** Professor N.K.U.A.

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μοντέλα Φωτισμού Βασισμένα στη Φυσική για Μηχανές Γραφικών Πραγματικού Χρόνου

Κωνσταντίνος Α. Κυριακός

Α.Μ.: 1115201600079

ΕΠΙΒΛΕΠΩΝ: **Θεοχάρης Θεοχάρης,** Καθηγητής Ε.Κ.Π.Α.

ABSTRACT

The subject of this thesis pertains to the physically based lighting models for real time game engines. Over the last decades, we have seen a great improvement in the lighting models, however, the dilemma between a good visual result and better performance still exists and has many variations depending on the hardware that will render the graphics.

In chapter 1, we see the most common lighting models which have been used over the last decades on real time game engines.

In chapter 2, we examine the characteristics that a model should have in order to be physically based. Moreover, we analyze the most popular physically based lighting models, along with their differences.

In chapter 3, we implement the lighting models in the Unity game engine and we compare the visual result and the performance of each one. Lastly, our comments are presented based on the comparisons made.

SUBJECT AREA: Computer Graphics
KEYWORDS: physically based rendering, graphics engine, shader programming

ΠΕΡΙΛΗΨΗ

Το αντικείμενο της πτυχιακής εργασίας αφορά τα ρεαλιστικά μοντέλα φωτισμού σε μηχανές γραφικών πραγματικού χρόνου. Τις τελευταίες δεκαετίες έχει γίνει σημαντική πρόοδος στα μοντέλα φωτισμού, ωστόσο το πρόβλημα της εξισορρόπησης μεταξύ του καλύτερου οπτικού αποτελέσματος και της καλύτερης επίδοσης παραμένει και ποικίλει ανάλογα το τελικό υλικό υπολογιστή που θα κληθεί να σχεδιαγραφήσει τα γραφικά.

Στο κεφάλαιο 1 θα δούμε τα πιο βασικά μοντέλα φωτισμού που χρησιμοποιήθηκαν τις τελευταίες δεκαετίες και χρησιμοποιούνται μέχρι και σήμερα σε μηχανές γραφικών πραγματικού χρόνου.

Στο κεφάλαιο 2 θα δούμε τι χαρακτηριστικά έχει ένα μοντέλο φυσικού φωτισμού και ποιες προϋποθέσεις πρέπει να πληροί για να θεωρηθεί μοντέλο φυσικού φωτισμού. Ακόμα, θα δούμε τα πιο γνωστά μοντέλα φυσικού φωτισμού, καθώς και τις μεταξύ τους διαφορές.

Στο κεφάλαιο 3 υλοποιήσαμε τα μοντέλα στη μηχανή γραφικών Unity και συγκρίνουμε τόσο το οπτικό αποτέλεσμα όσο και τις επιδόσεις του κάθε μοντέλου. Τέλος, παραθέτουμε τα συμπεράσματα μας βάση των συγκρίσεων που κάναμε.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Γραφικά Υπολογιστών
ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: σχεδιαγράφηση γραφικών βασισμένα στη φυσική, μηχανές γραφικών, προγραμματισμός φωτοσκιαστών

TABLE OF CONTENTS

TABLE OF FIGURES

SUMMARY OF SYMBOLS

Symbol	Definition
f	BRDF
f_d	Diffuse component of BRDF
f_r	Specular component of BRDF
ω_i	Incoming light vector
ω_o	Perfectly reflected light
ω_r	View light vector
ω_l	Light vector from the point on the surface toward the light source
L_i	Incident radiance
L_r	Reflected radiance
a	Shininess of material
n	Surface normal vector
φ	Azimuth angle
θ_i	Angle between ω_i and n
θ_r	Angle between ω_r and n
θ_h	Angle between h and n
h	Half unit vector between ω_i and ω_r
c	Color
$\chi^+(x)$	Equal to one if $x>0$ and zero if $x\leq 0$
k_a	Ambient reflection constant
k_d	Diffuse reflection constant
k_s	Specular reflection constant
Λ	Microfacet association function
n_{IOR}	Index of refraction

1. EARLY LIGHTING MODELS FOR REAL TIME GRAPHICS

1.1 Lambert lighting model

Lambert lighting model defines an ideal “matte” surface. A Lambertian surface reflects or emits equal (isotropic) luminance in every direction. For example, an evenly illuminated diffuse flat surface, like a piece of paper, is approximately Lambertian, because the reflected radiance is the same in every direction from which you can see the surface of the paper. However, it does not have isotropic intensity, because the intensity varies according to the cosine law.

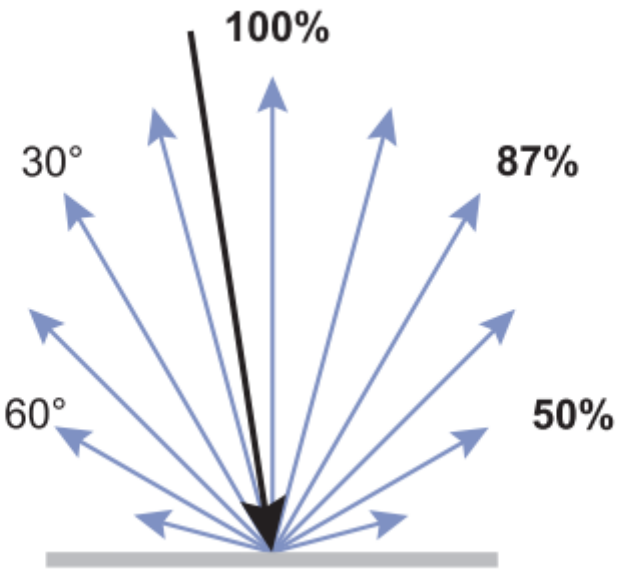

Figure 1: Lambertian surface [31]

According to Lambert’s cosine law the total reflected radiance observed by the viewer is directly proportional to the cosine of the angle θ_r between the direction of the viewer’s line of sight and the surface normal. In Figure 1 we see the reflected radiance intensity for a Lambertian surface. The formula that describes this behavior is:

$$L_r = L_i \cos(\theta_r)$$

In Figure 2, the visual result of Lambert’s lighting model is presented.

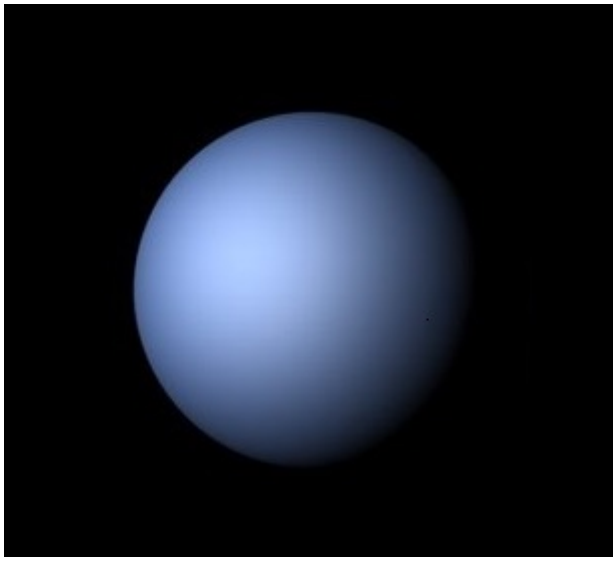

Figure 2: Lambert Lighting [19]

1.2 Phong lighting model

Phong lighting model may not be strictly part of the physical based lighting models but it deserves to be noticed due to the strong impact and influence that had in the field of graphics. Phong model was widely used for many decades and is still used because of its low computational cost for a quite good visual result.

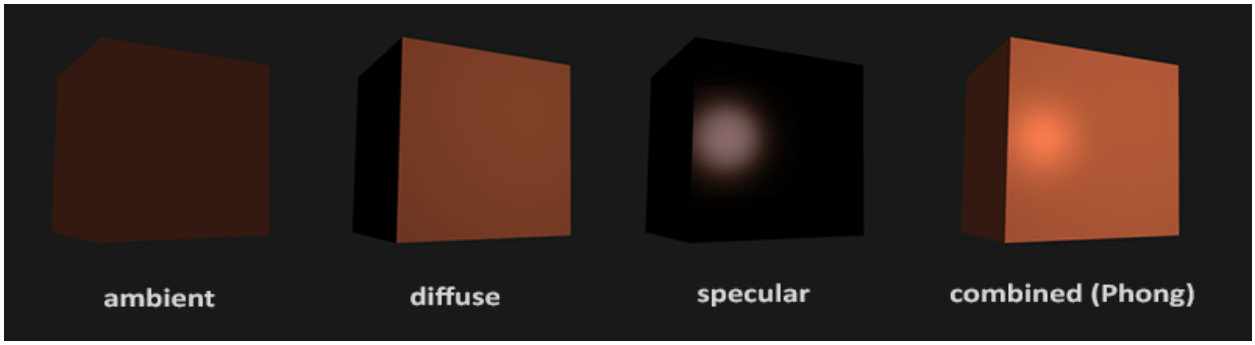

Figure 3: Phong lighting components [20]

The Phong illumination model is a combination of three components, as shown in Figure 3. The ambient reflection component which simulates the environment radiance that is reflected from other objects or distant light sources, like the moon. The diffuse reflection component simulates the radiance that is scattered uniformly when it strikes the object's surface. Lastly, the specular reflection component simulates the bright spots appearing on smooth shiny surfaces.

$$f = \underbrace{I_a k_a}_{\text{ambient component}} + \underbrace{I_j k_d (n \cdot \omega_l)}_{\text{diffuse component}} + \underbrace{I_j k_s (\omega_o \cdot \omega_r)^a}_{\text{specular component}},$$

Where $0 \le I_a$ is the ambient light intensity, $I_j \le 1$ is the light intensity and $0 \le k_a, k_d, k_s \le 1$ are constants which represent the ratio of reflection of each component.

1.3 Blinn-Phong lighting model

Blinn-Phong lighting model is basically a modified version of Phong lighting model. Blinn simplified the calculation of Phong's model by using the bisector vector h of l, ω_r :

$$\underbrace{I_j k_s (n \cdot h)^a}_{specular\ component}, \quad where\ h = \frac{\omega_i + v}{\left\| \omega_i + v \right\|}$$

Because of the low computational complexity without loss in the visual result, Blinn-Phong model has been broadly used in real time game engines and has been the core shading model until Direct3D 10 and OpenGL 3.1.

1.4 Normalization of Blinn-Phong lighting model

An important improvement in the Blinn-Phong model is the normalized version of it. Normalization means the scale of specular highlight, so that the reflected energy remains constant for varying surface smoothness. Normalization is a way to simulate energy conservation and achieve more realistic lighting. The difference between normalization and energy conservation is that energy conservation is strictly bounded for all incoming light, while normalization is as strong as we define it. This means that it is possible to normalize a model without satisfying the energy conservation property, if the result is better in this situation. In the bibliography, we can find various normalization factors. A simple one with low computational cost is $\frac{\alpha + 8}{8\pi}$, so the Blinn-Phong would be described as:

$$f = \underbrace{I_a k_a}_{ambient\ component} + \underbrace{I_j k_d (n \cdot \omega_l)}_{diffuse\ component} + \underbrace{I_j k_s (n \cdot h)^a \frac{\alpha + 8}{8\pi}}_{specular\ component}, \quad where\ h = \frac{\omega_i + v}{\left\| \omega_i + v \right\|}$$

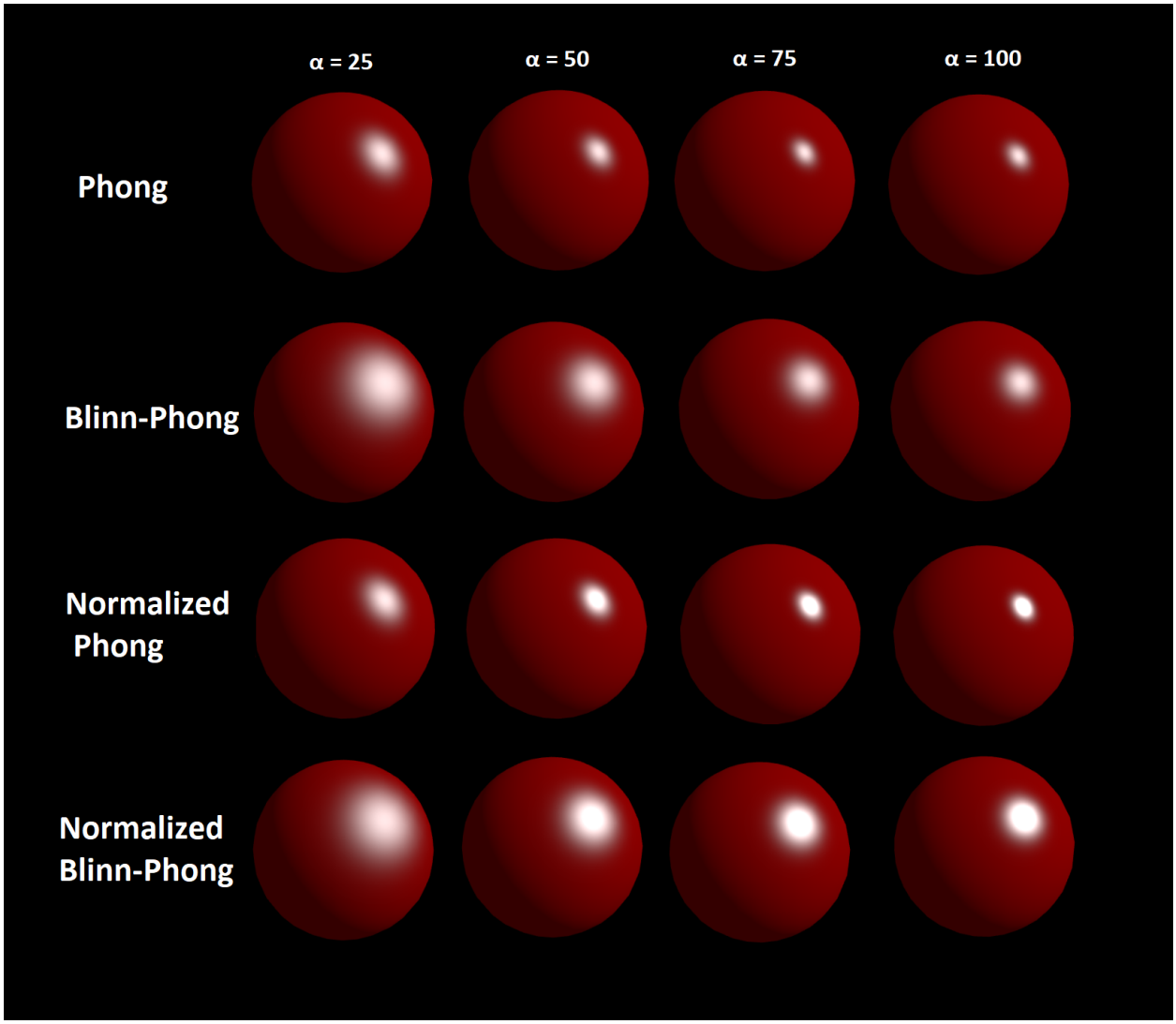

Figure 4: Red plastic balls rendered with Phong, Blinn-Phong and normalized factor for various smoothness values

The intent in Figure 4 is to render spheres made of the same material (red plastic), but with differing surface smoothness. It can be seen that in the two bottom rows, the highlight grows much brighter as it gets narrower, which is the correct behavior. The outgoing light is concentrated in a narrower cone. In the top two rows, the highlight remains equally bright as it narrows, so there is a loss of energy and surface reflectance appears to decrease.

2. PHYSICALLY BASED LIGHTING MODELS FOR REAL-TIME GRAPHICS

2.1 Physically based rendering theory

2.1.1 Bidirectional Reflectance Distribution Function (BRDF)

The BRDF was introduced by Nicodemus as a specific case of the BSSRDF (Bidirectional Scattering-Surface Reflectance-Distribution function) to describe how light is reflected at opaque surfaces. Contrary to BSSRDF, the BRDF does not take into account the subsurface scattering and assumes that light entering a material leaves the material at the same position (Figure 5).

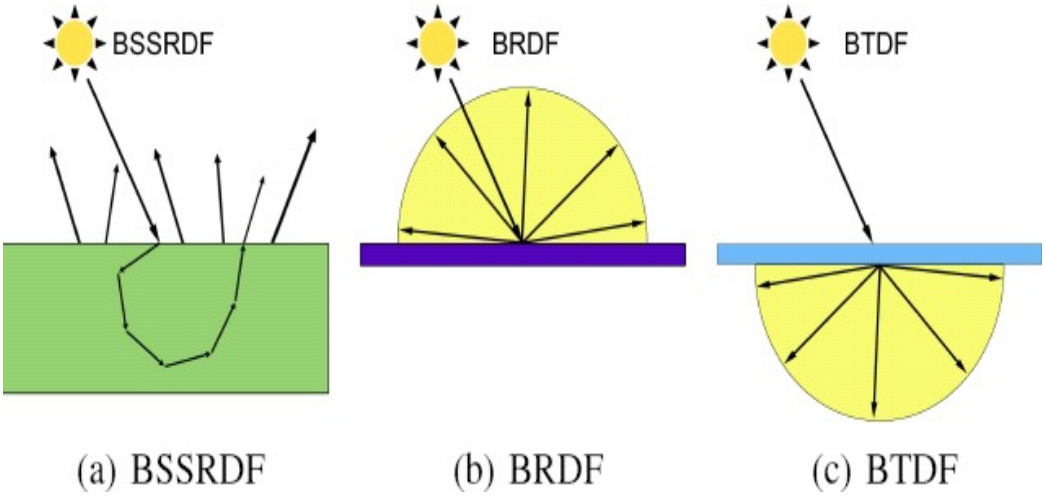

Figure 5: Scattering of light in (a) BSSRDF, (b) BRDF and (c) BTDF [7]

The BRDF is defined as the ratio of reflected radiance to incident irradiance:

$$f_r(\omega_i,\omega_r)=\frac{dL_r(\omega_r)}{dE_i(\omega_r)}=\frac{dL_r(\omega_r)}{dL_i(\omega_i)cos\theta_i d\omega_i}$$

where ω_i,ω_r are the incident and exitant direction of light, L_r is reflected radiance, E_i,L_i are the incident irradiance and incident radiance, accordingly and θ_i is the angle between ω_i and the surface normal.

2.1.2 Physically based BRDFs

A BRDF is considered to be physically based when it obeys the following physical properties:

- non-negativity
- Helmholtz reciprocity
- energy conservation

Non-negativity: the BRDF should not be negative for any pair of incoming and outgoing direction $f_r(\omega_i, \omega_r) \geq 0$

Helmholtz reciprocity: Helmholtz reciprocity principle states that the path of any pair of incoming and outgoing light is reversible $f_r(\omega_i, \omega_r) = f_r(\omega_r, \omega_i)$. However this principle is not always necessary but only for certain polarization states of light.

Energy conservation: Energy conservation assumes that the reflected energy from one point cannot be more than the incident energy the same point:

$$\forall \omega_i \in \Omega \qquad \int_{\omega_r \in \Omega} f_r(\omega_i, \omega_r) \cos\theta_r \, d\omega_r \leq 1 \quad where\ \Omega\ is\ the\ unit\ hemisphere$$

2.2 Physically based rendering models

2.2.1 Cook-Torrance lighting model

Based on geometrical optics theory and the studies of Beckman-Spizzichino and Torrance-Sparrow, Cook-Torrance developed a microfacet lighting model for rough surfaces that became popular in computer graphics and has been used widely until today. As Figures 6, and 7 show, microfacet BRDF states that surfaces are not smooth at a micro level, but they are made of a large number of randomly aligned planar surface fragments instead, which are called microfacets.

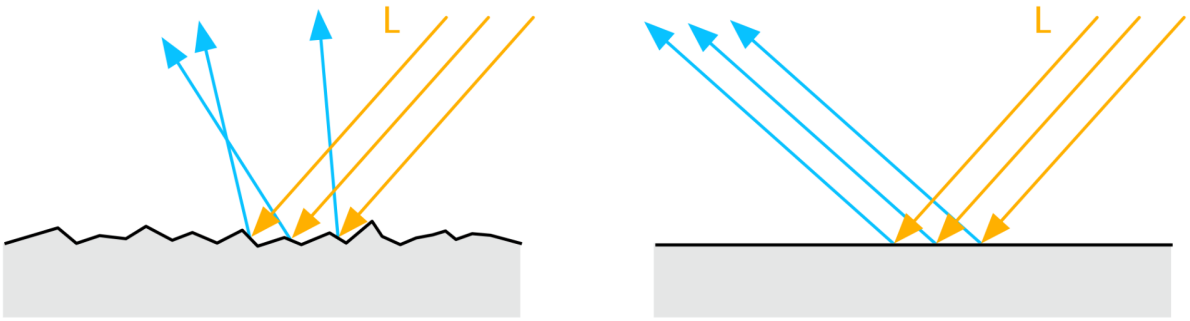

Figure 6: Surface as modeled by a microfacet model (left) and flat surface (right) [22]

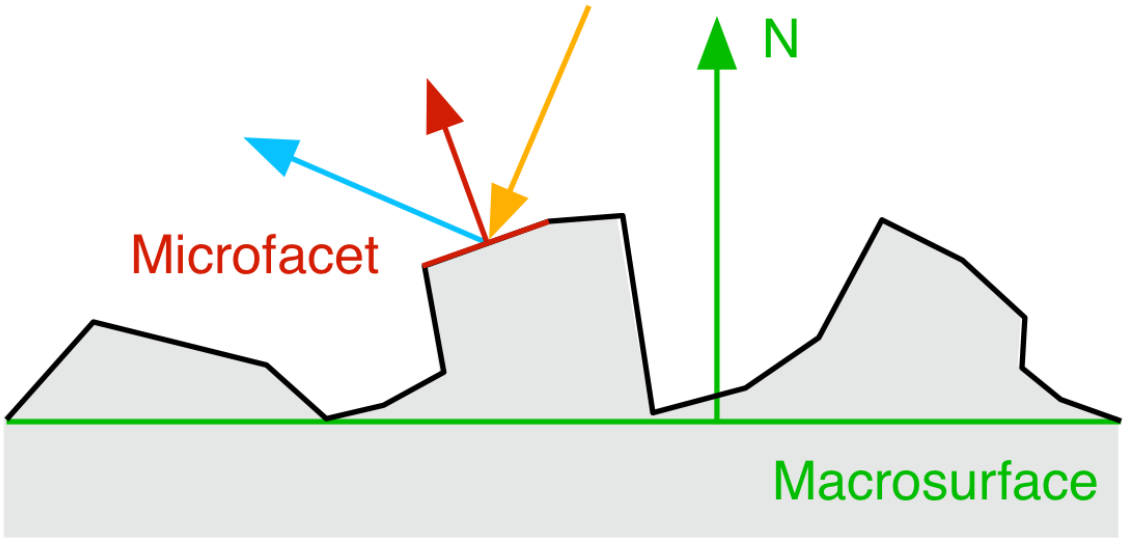

Figure 7: Microfacet at close view [22]

The model describes the reflectance as the linear combination of a diffuse and a specular component (Figure 8):

$$f_r = k_d f_d + k_s f_s, \qquad k_d + k_s = 1$$

where f_d describes the diffuse term, f_s the specular term and k_d, k_s are the diffuse and specular coefficients, accordingly. Practically k_d determines the amount of the incoming radiance that gets diffused and k_s is the amount of light that is specularly reflected.

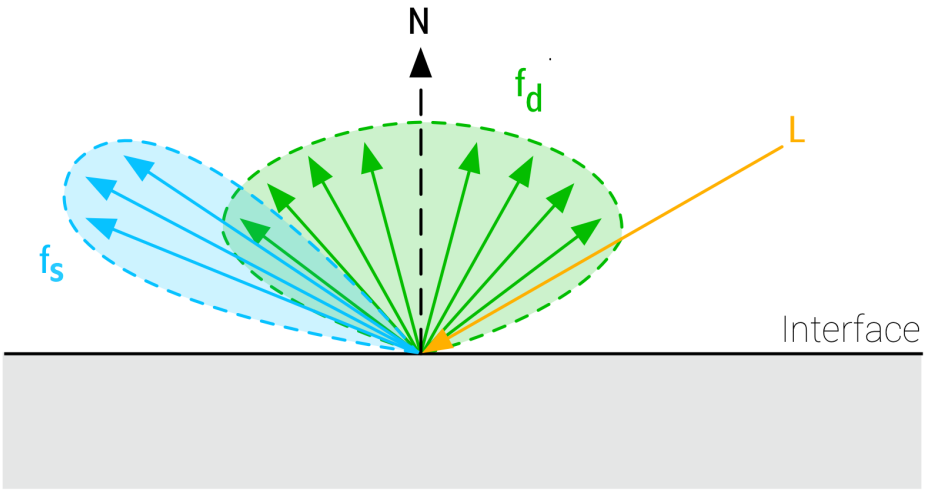

Figure 8: Representation of diffuse and specular term [22]

Through the k_d, k_s parameters the model introduced a new way to differentiate metallic (conductor) and non-metallic (dielectric) surfaces. A surface with high k_s will behave more like a mirror, while it will have high value of k_d , if it exhibits a strong diffusive behavior.

The diffuse term f_d is a Lambertian BRDF that ensures a uniform response over the microfacets hemisphere:

$$f_d = \frac{c}{\pi}$$, where c is the color.

Cook-Torrance assumes that only facets whose normal has the same direction with h contribute to the specular reflection.

The specular term f_s is composed of a normal distribution of microfacets D, a geometrical attenuation factor G for masking and shadowing between facets and a Fresnel term F which defines the ratio of reflected and transmitted energy at a point:

$$f_s = \frac{DGF}{\pi(\omega_i \cdot n)(\omega_r \cdot n)}$$

The D Term describes the fraction of facets that has the same orientation as the direction h . For the calculation of D Term Cook-Torrance uses the Beckmann distribution function because it is suitable for a wide range of materials and surface conditions:

$$D = \frac{e^{-[tan\theta_h / m]^2}}{\pi m^2 \cos^4 \theta_h} \quad , \quad \theta_h = \arccos(n \cdot h)$$

At a micro level perspective, there is a chance the incident or the outgoing light will be blocked by another facet. This effect is called shadowing-masking (Figure 9).

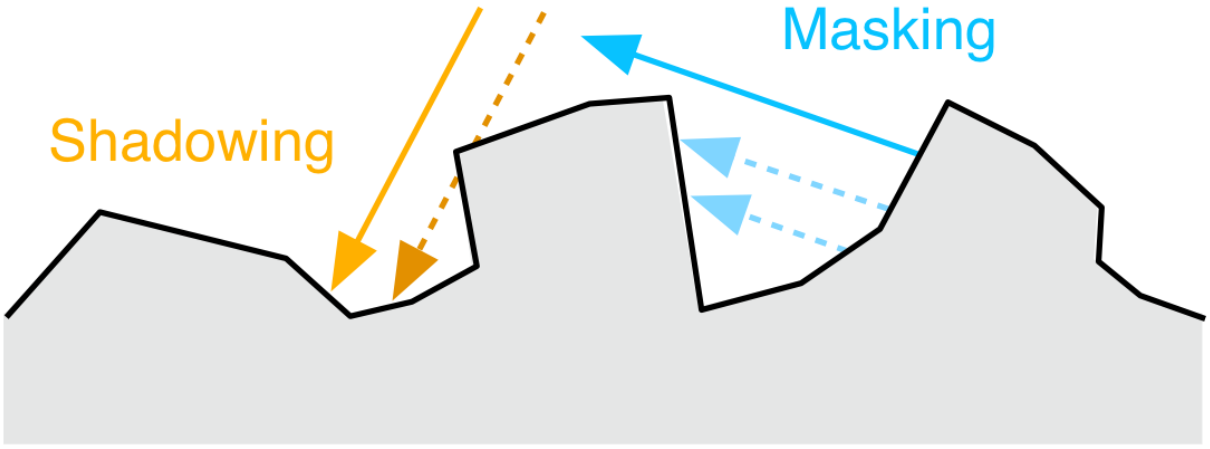

Figure 9: Masking and shadowing of microfacets [22]

The factor G models a shadowing-masking function. Cook-Torrance uses the Blinn's implementation who simplified the function based on Torrance-Sparrow's theory and the final expression is:

$$G = min\left\{1, \frac{2(n\cdot h)(n\cdot\omega_i)}{(\omega_r\cdot h)}, \frac{2(n\cdot h)(n\cdot\omega_r)}{(\omega_r\cdot h)}\right\}$$

The amount of light the viewer sees reflected from a surface depends on the viewing angle and index of refraction n_{IOR} of the material. When looking at the surface at normal incidence, the amount of light reflected is less than when looking at a grazing angle. This phenomenon is called the Fresnel effect and it is easily observable on large bodies of water, as shown in Figure 10.

Figure 10: The Fresnel effect on large volume of water [32]

The term F describes the Fresnel effect and the Fresnel equation for unpolarized incident light is:

$$F = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left(1+\frac{[c(g+c)-1]^2}{[c(g-c)+1]^2}\right) \text{ , where } c = \cos(\theta) = \omega_r\cdot h \text{ and } g^2 = n_{IOR}^2 + c^2 - 1.$$

Cook-Torrance suggests an approximation for the Fresnel term by interpolating the color of the material at $\theta = 0$ and the color at $\theta = \frac{\pi}{2}$, which is the color of the light source because $F_{\pi/2} = 1$ at every wavelength, so the equation for a color is:

$$c_i = \frac{1}{\pi}c_{i,0} + (c_{i,\pi/2} - c_{i,0})\frac{max(0, F_\theta(\lambda) - F_0(\lambda))}{F_{\pi/2} - F_0(\lambda)}, \text{ where } c_i = [\text{red}, \text{blue}, \text{green}]$$

2.2.2 Schlick's approximation of the Fresnel Factor

The Fresnel approximation that Cook-Torrance introduced in their model is quite computationally expensive for real time rendering. Schlick, using the method of rational fraction approximation, created an approximation for the Fresnel factor that is almost 32 times faster to calculate and has less than 1% error in comparison with the true Fresnel factor. The final function is:

$$F_{Schlick} = f_0 + (f_{90} - f_0)(1 - \omega_r \cdot h)^5, where\ \ f_o = \left(\frac{n_{IOR} - 1}{n_{IOR} + 1}\right)^2$$

The constant f_0 represents the specular reflectance at normal incidence and is achromatic for dielectrics, and chromatic for metals. The actual value depends on the index of refraction (IOR) of the interface. This Fresnel function can be seen as interpolating between the incident specular reflectance and the reflectance at grazing angles, represented here by f_{90} . Observation of real world materials shows that both dielectrics and conductors exhibit achromatic specular reflectance at grazing angles and that the Fresnel reflectance is 1.0 at 90°. Schlick's approximation has been a great addition in the physically based render and is still used by every modern graphics engine.

2.2.3 Oren-Nayar lighting model

Oren-Nayar enhanced the Lambertian model for rough diffuse surfaces in order to describe, in a more realistic way, the behaviour of real-world materials like concrete, sand and cloth, which exhibit increasing brightness as the viewing direction approaches the light source direction, rather than being independent of the viewing direction. This model is able to explain the view dependence appearance of the matte surfaces with geometric optics. A rough diffuse surface is modelled as a collection of long symmetric V-cavities, each of which consists of two microfacets with a Lambertian reflectance. Microfacets orientated towards the light source, reflect diffusely some light back to the light source (backscatter). Given an incident direction, the total reflected radiance is the integral of grooves' reflectance values, the projected reflected radiance and the radiance due to the bounces between them. Oren & Nayar found hard to solve the resulting equations analytically, so they presented a functional approximation using radiometric terms following Figure 11:

$$f_r(\omega_i, \omega_r) = \frac{\rho}{\pi}(A + B\,max(0, \cos(\varphi_i - \varphi_r))\sin(b_1)\cdot\sin(b_2)),\ \ \rho = albedo$$

$$where\ \ b_1 = max(\theta_i, \theta_r),\ \ b_2 = min(\theta_i, \theta_r),\ \ A = 1 - 0.5\frac{\alpha^2}{a^2 + 0.33},\ \ B = 0.45\frac{\alpha^2}{a^2 + 0.09}$$

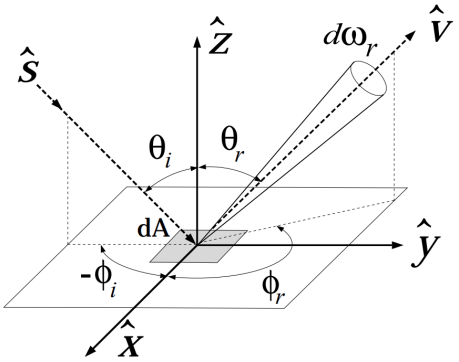

Figure 11: Geometry used to define radiometric terms [13]

As we can see in the Figure 12, a diffuse surface like the full moon does not follow the Lambertian model. The moon is lit very evenly and, viewed from Earth, it appears to have a smooth(ish) surface. However, the light does not fall off around the edges in a consistent with Lambert's model way. This model, widely used in computer graphics, obeys the reciprocity principle and reduces to the Lambertian model when $a=0$.

Figure 12: Lambert diffuse in comparison with the Oren-Nayar diffuse and full moon [23]

2.2.4 Walter et al. lighting model

The Walter et al. lighting model extends the microfacet theory of Cook-Torrance by simulating the transmission of light through rough surfaces making a BSDF (Bidirectional Scattering Distribution Function).The most interesting is the BRDF part of the model for which they created a new microfacet distribution function for the D term named GGX. In combination with Smith's approximation for the shadowing-masking term G achieves brighter highlights and longer tails (Figure 13) than the previous most used distributions, such as Phong and Beckmann, contributing to more realistic lighting.

Figure 13: Comparison between GGX (left) and Normalized Phong (right) [24]

As a result, the GGX distribution has been adopted widely in recent years by real-time graphics engines, taking the place of the normalized Blinn-Phong. GGX distribution is:

$$D(h)=\frac{a_g^2\chi^+(h\cdot n)}{\pi cos^4\theta_h(a_g^2+\tan^2\theta_h)^2},\ where\ a_g^2\ is\ a\ width\ parameter$$

Smith's approximation G for shadowing-masking is given as the product of two monodirectional shadowing terms G_1 :

$$G(\omega_i,\omega_r,h)\approx G_1(\omega_i,h)G_1(\omega_r,h)$$

Where G_1 is:

$$G_1(\omega_x,h)=\chi^+\left(\frac{\omega_x\cdot h}{\omega_x\cdot n}\right)\frac{2}{1+\sqrt{1+a_g^2\tan^2(\theta_x)}}\quad ,where\ \ \omega_x\in(\omega_i,\omega_r),\ \ \theta_x\in(\theta_i,\theta_r)$$

2.2.5 Eric Heitz validation and improvement on Smith's G term

Smith's masking function is expressed as an integral over the slopes of the microsurface and its form is derived with a ray-tracing formulation of the masking probability. As a result, this makes the Smith masking function to be considered an approximate solution. On the other hand, the masking function of Ashikhmin et al. emphasizes on the correctness. Eric Heitz showed that the derivation of Ashikhmin

masking function leads to the same result as the generalized form of Smith's masking function which proved that Smith's masking function is exact.

Heitz notices also that the Smith's masking function for the G term, calculates the shadowing and masking separately and multiplies them in the end. However, some correlations always exist between the microfacets, so Smith's function overestimates the shadowing. As we can see practically in Figure 14, Smith's G distributes the light among the normals which are hit directly by the light source and the same normals in the shadow. This distribution has, as a result, the reduction of outgoing light from higher normals.

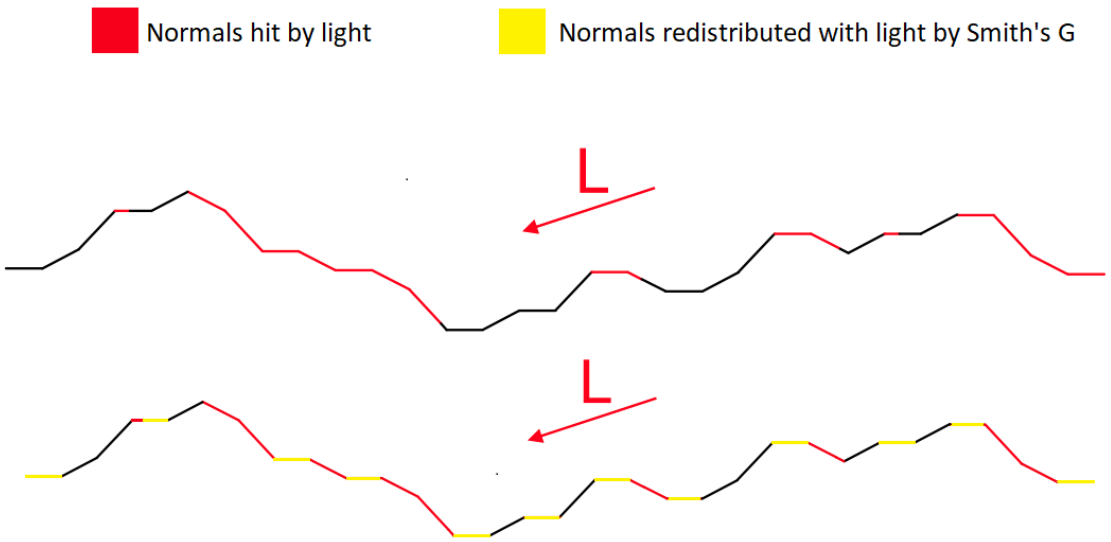

Figure 14: Uncorrelated G (Smith) light redistribution [33]

The more a microfacet is elevated within the microsurface, the higher the probability of it being visible for the incident/outgoing directions. Taking this into consideration, Heitz introduced a height-correlated G function:

$$G(\omega_i,\omega_r,h)=\frac{\chi^+(\omega_r\cdot h)}{1+\Lambda(\omega_r)}\frac{\chi^+(\omega_i\cdot h)}{1+\Lambda(\omega_i)}=,where\ \ \Lambda(\omega_x)=\frac{-1+\sqrt{1+a^2\tan^2(\theta_x)}}{2}$$

The visual results of the height-correlated G can be seen in Figure 15. Correlated G achieves brighter highlights with less energy loss, so the final lighting is more realistic.

Figure 15: Comparison between height correlated G(Heitz) and uncorrelated G(Smith) [33]

2.2.6 Disney lighting model contribution in real time graphics

Disney's lighting model (also known as Burley model) is a film production lighting model which Disney used in many movies, accomplishing good visual results. Even though the model was created for movies, with the evolution of hardware, many modern graphics engines picked up the ideas that can work in a real time graphics pipeline. The biggest interest can be found in the diffuse part of the model. Disney evaluated Lambert's diffuse model but found that it is often too dark on the edges and by adding a Fresnel factor to make it more physically plausible, only makes it darker. Through observation, Disney created an empirical model for diffuse retroreflection that transitions between a diffuse Fresnel shadow for smooth surfaces and an added highlight for rough surfaces. So, by modifying the Fresnel factor to return a value for grazing angle based on roughness rather than zero, they ended up with the following formula for the diffuse term:

$$f_d = \frac{c}{\pi} F_{Schlick}(n, \omega_i, 1, f_{90}) F_{Schlick}(n, \omega_r, 1, f_{90})$$

$$f_d = \frac{c}{\pi}(1+(f_{90}-1)(1-n\cdot\omega_i)^5)(1+(f_{90}-1)(1-n\cdot\omega_r)^5)$$

$$where\ \ f_{90} = 0.5+2a(\omega_i\cdot h)^2$$

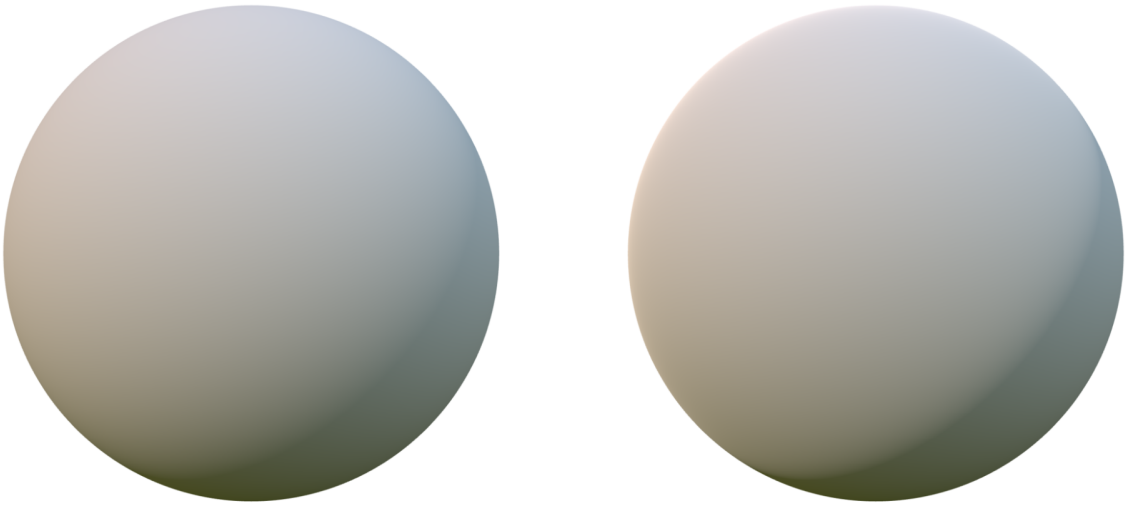

Figure 16: Comparison between the Lambertian diffuse BRDF (left) and the Disney diffuse BRDF (right) [15]

As we can see in Figure 16, the edge of the ball in the Disney's diffuse model is much more bright and approximates better real world lighting.

3. COMPARISON AND CONCLUSION

3.1 Lighting models comparison overview

The lighting models that we have seen can be categorized in 3 groups. Lighting models for diffuse reflection, empirical lighting models, physically based lighting models. The lighting models for the diffuse reflection are the Lambert, Oren-Nayar and Disney. The empirical lighting models are the Phong, Blinn-Phong and their normalized versions. Lastly, the physically based lighting models are all based on the microfacet theory of Cook-Torrance and their difference lies in how the terms D, G, F are calculated. We chose the most widely used combinations. For the Fresnel term F, Schlick's approximation has been established as the go-to solution, because of the low computational cost in combination with the quite accurate visual result, so we use it in every model. The models that we are going to compare are the Cook-Torrance (F=Schlick), Walter (F=Schlick) and Walter with Heitz's height correlated masking function Walter (G=Heitz, F=Schlick).

3.2 Visual Comparison

To compare the visual difference of each lighting model we programmed the HLSL shaders in Unity engine and applied them on the same robot model. In order to keep the points of difference as accurate as possible, we used the same color and variant values wherever it was applicable.

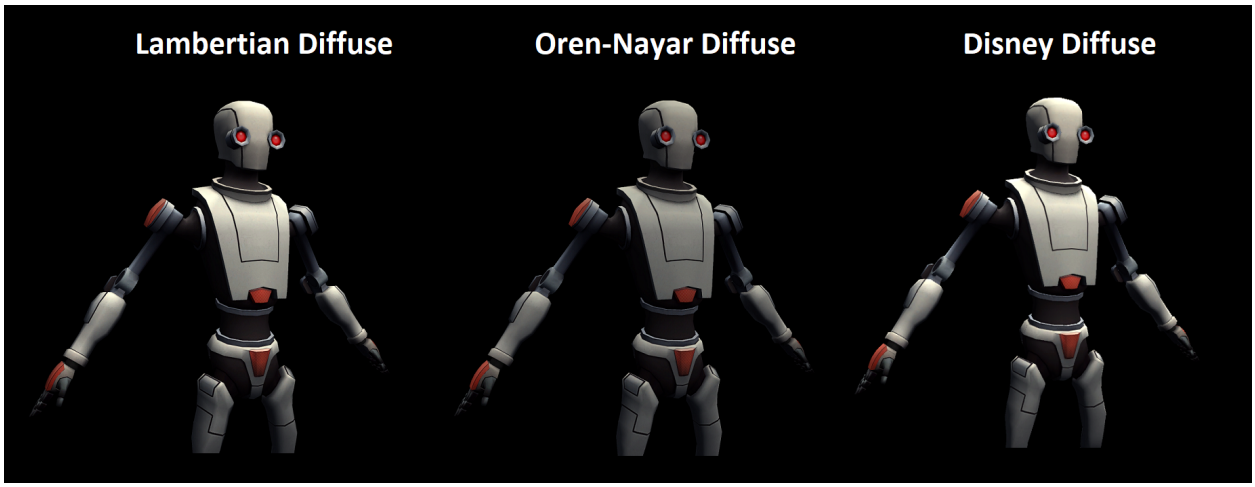

Figure 17: Lambert, Oren-Nayar and Bulrey visual comparison

In Figure 17, we see the difference of diffuse models and how they improved through the passage of time. Lambert diffuse has a big contrast between the highlighted and shadowed parts of the model. Oren-Nayar diffuse accomplishes milder highlights leading to a more uniform visual result. And lastly, Burley has similar visual result to Lambert, but with some additional light in the edges because of the Fresnel term.

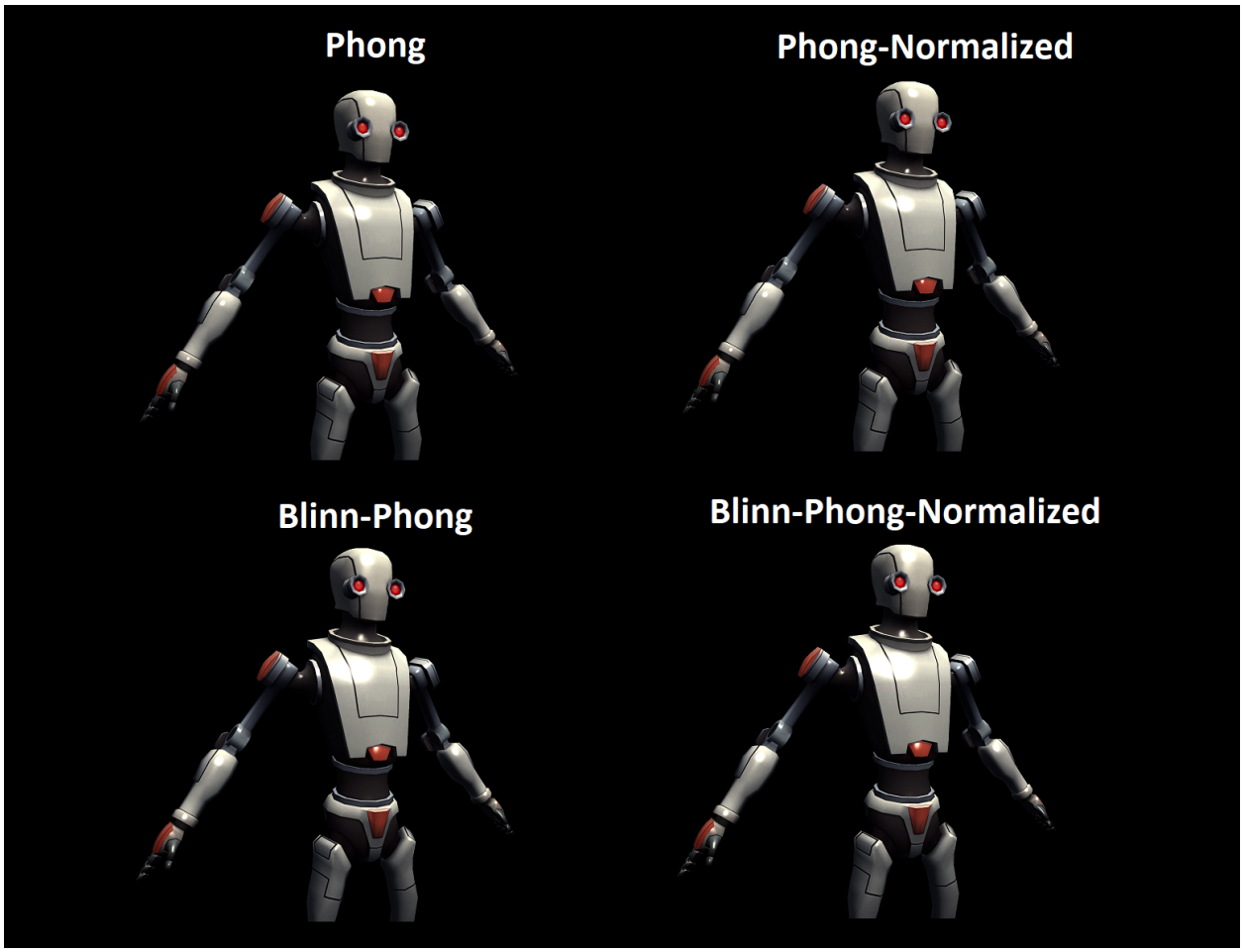

Figure 18: Phong, Phong-Normalized, Blinn-Phong and Blinn-Phong normalized visual comparison

In Figure 18, we see the Phong and Blinn-Phong lighting models with their normalized variations. Even though these lighting models are simplistic, they achieve a quite good visual result. Blinn-Phong has brighter highlights and the normalized versions have even brighter highlights for both models. However, the shadowed areas are little to completely dark and, thus, the viewer ends up losing precious information and details of the lighted object. In Figure 18, this phenomenon is noticeable at the joints of legs, the neck, under the shoulders etc., where these body parts look disjointed from the rest of the body.

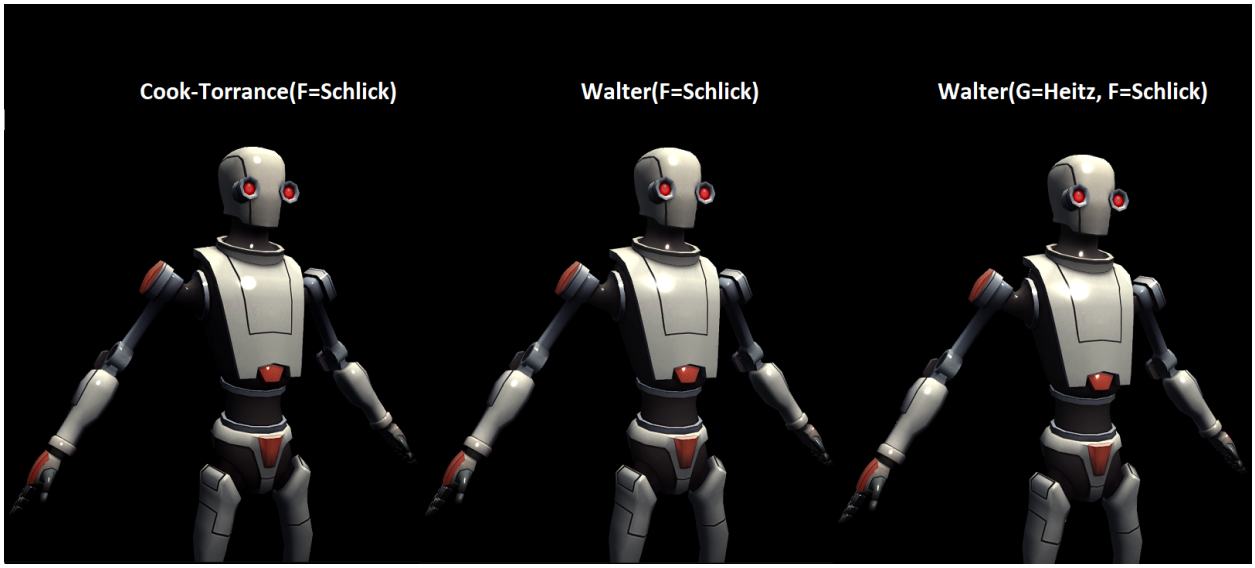

Figure 19: Cook-Torrance variations visual comparison

In Figure 19, we see the Cook-Torrance physically based lighting model with some popular variations. The basic Cook-Torrance variation with the Beckmann distribution function has a really good visual result. The diffuse reflection is better distributed in the parts of the body in comparison with the previous models. The darker areas such as the side of the head and the low part of the arm are brighter and, so, they are more easily distinguished from the background color. The glossiness that we expect from a metallic robot is well displayed and in general the final result looks more realistic and detailed, even though the sample model is the same. The second variation with the GGX distribution function is brighter with longer tails on the specular reflection which gives a more realistic feeling. Lastly, the Height-Correlated GGX distribution function is even brighter than GGX and adds some small reflections on the edges reducing the lost energy.

3.3 Performance Comparison

To measure the performance of each lighting model we implemented a benchmark test in Unity engine for each model. The test spawns eternally game objects which are destroyed after 2 seconds so we do not get a memory throttling. In order to get the most accurate results, we reduced the GPU clocks to keep the GPU temperature stable and, in consequence, the clock's frequency as well. As seen in Figure 20 we used three metrics to measure the performance of each model, the 0.1% low FPS, the 1% low FPS and the average FPS. The 0.1%/1% low FPS is a good metric to measure the consistency of frame rate. The time each frame needs to be rendered can be transformed to FPS by dividing one second with it. The 0.1%/1% low FPS are the 0.1%/1% slowest frame render times averaged and transformed to FPS. The 0.1%/1% low FPS is a good way to express the possibility of image stuttering, and even if a game has high average FPS, a smooth viewing experience can not be guaranteed.

Figure 20: Benchmark results showing the 0.1% low, 1% low and average frames per second

By looking at the benchmark results, we can distinguish three main categories: the old empirical models (Phong, Blinn-Phong), the diffuses (Lambertian, Oren-Nayar, Burley) and the physically based models (Cook-Torrance and its variations). Between the physically based models and the empirical models, we observe about 40% difference in performance, which is expected, when taking into account the massive improvement in the visual result. Within each category the performance gap is reduced to around 1% which is a small yet considerable amount, if you need to squish as much performance as possible from the hardware.

3.4 Conclusion

Over the last decades, the field of graphics has seen great improvement. The models we examine in this thesis are constantly evaluated and improved. Also, we have to bear in mind that there is not an all-in-one model that works for everyone. Depending on the needs of each creator and the consumer, the choice should be different. If we want to create a game engine for mobile devices, we should pay more attention to the performance, rather than to the best visual result. Even for mobile devices, the choice depends on the target market. For low/mid-range mobile devices a Phong variation would be the best choice. But, in case the aim is only the high end mobile devices, Cook-Torrance (D=HCGGX) in combination with Lambert diffuse is a viable combination that achieves the realistic visual result of physically based rendering with the highest possible performance. On the other hand, if the target market is the desktop users, the art department can make any choice that works better for them, such as the Cook-

Torrance (D=GGX) with Oren-Nayar diffuse. In conclusion, the choice of the lighting model is not a one sided problem; many parameters should be taken into account and different models work better for each case.

4. APPENDIX

4.1 Lambert shader

```
Shader "Custom/lambert"
{
    Properties{
        _Color("Color", Color) = (1.0,1.0,1.0)
        _Tex ("Pattern", 2D) = "white" {}
    }

    SubShader{
        Tags { "RenderType" = "Opaque" }
        LOD 200 //Level of detail

        Pass{
            Tags {"LightMode" = "ForwardBase"}
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag
            #pragma target 3.0

            #include "UnityCG.cginc"
            // user defined variables
            uniform float4 _Color;

            sampler2D _Tex; //Used for texture
            float4 _Tex_ST; //For tiling

            // unity defined variables
            uniform float4 _LightColor0;

            // base input structs
            struct vertexInput {
                float4 vertex: POSITION;
                float3 normal: NORMAL;
                float2 uv : TEXCOORD0;
            };

            struct vertexOutput {
                float4 pos: SV_POSITION;
                float4 col: COLOR;
                float2 uv : TEXCOORD0;
            };


            // vertex functions
            vertexOutput vert(vertexInput v) {
                vertexOutput o;
```

```
                half3 normalDirection =
normalize(mul(v.normal,unity_WorldToObject).xyz);
                half3 lightDirection = normalize(_WorldSpaceLightPos0.xyz);

                half NdotL = saturate(dot(normalDirection, lightDirection));
                fixed3 diffuseReflection = _LightColor0.xyz * _Color.rgb * NdotL;
                fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.rgb* _Color.rgb;


                o.col = fixed4(diffuseReflection + ambient, 1.0);
                o.pos = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _Tex);
                return o;
            }

            // fragment function
            float4 frag(vertexOutput i) : COLOR
            {
                return i.col*tex2D(_Tex, i.uv);
            }
        ENDCG
        }
    }
}
```

4.2 Phong shader [34]

```
Shader "Custom/phong"
{
Properties {
      _Color ("Color", Color) = (1, 1, 1, 1) //The color of our object
      _Tex ("Pattern", 2D) = "white" {} //Optional texture

      _Shininess ("Shininess", Float) = 10 //Shininess
      _SpecColor ("Specular Color", Color) = (1, 1, 1, 1) //Specular highlights color
   }
   SubShader {
      Tags { "RenderType" = "Opaque" } //We're not rendering any transparent objects
      LOD 200 //Level of detail

      Pass {
         Tags { "LightMode" = "ForwardBase" } //For the first light

         CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma target 3.0

               #include "UnityCG.cginc" //Provides us with light data, camera information,
etc
```

```
            uniform float4 _LightColor0; //From UnityCG

            sampler2D _Tex; //Used for texture
            float4 _Tex_ST; //For tiling

            uniform float4 _Color; //Use the above variables in here
            uniform float4 _SpecColor;
            uniform float _Shininess;

            struct appdata
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float4 pos : POSITION;
                float3 normal : NORMAL;
                float2 uv : TEXCOORD0;
                float4 posWorld : TEXCOORD1;
            };

            v2f vert(appdata v)
            {
                v2f o;

                    o.posWorld = mul(unity_ObjectToWorld, v.vertex); //Calculate the world
position for our point
                 o.normal = normalize(mul(float4(v.normal, 0.0), unity_WorldToObject).xyz);
//Calculate the normal
                o.pos = UnityObjectToClipPos(v.vertex); //And the position
                o.uv = TRANSFORM_TEX(v.uv, _Tex);

                return o;
            }

            fixed4 frag(v2f i) : COLOR
            {
                float3 normalDirection = normalize(i.normal);
                            float3 viewDirection = normalize(_WorldSpaceCameraPos -
i.posWorld.xyz);

                float3 vert2LightSource = _WorldSpaceLightPos0.xyz - i.posWorld.xyz;
                float oneOverDistance = 1.0 / length(vert2LightSource);
                      float3 lightDirection = _WorldSpaceLightPos0.xyz - i.posWorld.xyz *
_WorldSpaceLightPos0.w;
             //Ambient component
                          float3 ambientLighting = UNITY_LIGHTMODEL_AMBIENT.rgb *
_Color.rgb;
```

```
                    float3 diffuseReflection = _LightColor0.rgb * _Color.rgb * max(0.0, dot(normalDirection, lightDirection)); //Diffuse component
                float3 specularReflection;
                if (dot(i.normal, lightDirection) < 0.0) //Light on the wrong side - no specular
                {
                    specularReflection = float3(0.0, 0.0, 0.0);
                }
                else
                {
                    //Specular component
                     specularReflection = _LightColor0.rgb * _SpecColor.rgb * pow(max(0.0, dot(reflect(-lightDirection, normalDirection), viewDirection)), _Shininess);
                }

                    float3 color = (ambientLighting + diffuseReflection) * tex2D(_Tex, i.uv) + specularReflection; //Texture is not applient on specularReflection
                return float4(color, 1.0);
            }
        ENDCG
    }
  }
}
```

4.3 Blinn-Phong Shader

```
Shader "Custom/blinn_phong"
{
Properties {
    _Color ("Color", Color) = (1, 1, 1, 1) //The color of our object
    _Tex ("Pattern", 2D) = "white" {} //Optional texture

    _Shininess ("Shininess", Float) = 10 //Shininess
    _SpecColor ("Specular Color", Color) = (1, 1, 1, 1) //Specular highlights color
  }
  SubShader {
    Tags { "RenderType" = "Opaque" } //We're not rendering any transparent objects
    LOD 200 //Level of detail

    Pass {
      Tags { "LightMode" = "ForwardBase" } //For the first light

      CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #pragma target 3.0
          #include "UnityCG.cginc" //Provides us with light data, camera information, etc

        uniform float4 _LightColor0; //From UnityCG

        sampler2D _Tex; //Used for texture
        float4 _Tex_ST; //For tiling
```

```
            uniform float4 _Color; //Use the above variables in here
            uniform float4 _SpecColor;
            uniform float _Shininess;

            struct appdata
            {
               float4 vertex : POSITION;
               float3 normal : NORMAL;
               float2 uv : TEXCOORD0;
            };

            struct v2f
            {
               float4 pos : POSITION;
               float3 normal : NORMAL;
               float2 uv : TEXCOORD0;
               float4 posWorld : TEXCOORD1;
            };

            v2f vert(appdata v)
            {
               v2f o;

                  o.posWorld = mul(unity_ObjectToWorld, v.vertex); //Calculate the world
position for our point
                o.normal = normalize(mul(float4(v.normal, 0.0), unity_WorldToObject).xyz);
//Calculate the normal
               o.pos = UnityObjectToClipPos(v.vertex); //And the position
               o.uv = TRANSFORM_TEX(v.uv, _Tex);

               return o;
            }

            fixed4 frag(v2f i) : COLOR
            {
               float3 normalDirection = normalize(i.normal);
                        float3 viewDirection = normalize(_WorldSpaceCameraPos -
i.posWorld.xyz);

               float3 vert2LightSource = _WorldSpaceLightPos0.xyz - i.posWorld.xyz;
               float oneOverDistance = 1.0 / length(vert2LightSource);

                     float3 lightDirection = _WorldSpaceLightPos0.xyz - i.posWorld.xyz *
_WorldSpaceLightPos0.w;
               float3 reflectDirenction = reflect(-lightDirection, normalDirection);
               float3 halfDirection = normalize(lightDirection + viewDirection );
                 //Ambient component
                        float3 ambientLighting = UNITY_LIGHTMODEL_AMBIENT.rgb *
_Color.rgb;
                     float3 diffuseReflection = _LightColor0.rgb * _Color.rgb * max(0.0,
dot(normalDirection, lightDirection)); //Diffuse component
```

```
                float3 specularReflection;
                if (dot(i.normal, lightDirection) < 0.0) //Light on the wrong side - no specular
                {
                    specularReflection = float3(0.0, 0.0, 0.0);
                }
                else
                {
                    //Specular component
                     specularReflection = /*attenuation * */_LightColor0.rgb * _SpecColor.rgb
* pow(max(0.0, dot(halfDirection, normalDirection)), _Shininess);
                }

                    float3 color = (ambientLighting + diffuseReflection) * tex2D(_Tex, i.uv) +
specularReflection; //Texture is not applient on specularReflection
                return float4(color, 1.0);
            }
        ENDCG
    }
  }
}
```

4.4 Cook-Torrance (F = Schlick)

```
Shader "Custom/ct_beck_schlick" {
    Properties {
      _Color ("Color", Color) = (1, 1, 1, 1) //The color of our object
      _Tex ("Pattern", 2D) = "white" {} //Optional texture

      _Roughness ("Roughness", Range(0.000000001,1)) = 0.5
      _Fresnel("Fresnel Value", Float) = 0.028
      _SpecColor ("Specular Color", Color) = (1, 1, 1, 1) //Specular highlights color
   }
   SubShader {
      Tags { "RenderType" = "Opaque" } //We're not rendering any transparent objects
      LOD 200 //Level of detail


      Pass {
         Tags { "LightMode" = "ForwardBase" } //For the first light

         CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma target 3.0

               #include "UnityCG.cginc" //Provides us with light data, camera information,
etc

            uniform float4 _LightColor0; //From UnityCG

            sampler2D _Tex; //Used for texture
            float4 _Tex_ST; //For tiling
```

```
            uniform float4 _Color; //Use the above variables in here
            uniform float4 _SpecColor;
            uniform float _Fresnel, _Metallic, _Roughness;

            struct appdata
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float4 pos : POSITION;
                float3 normal : NORMAL;
                float2 uv : TEXCOORD0;
                float4 posWorld : TEXCOORD1;
            };

            v2f vert(appdata v)
            {
                v2f o;

                    o.posWorld = mul(unity_ObjectToWorld, v.vertex); //Calculate the world position for our point
                 o.normal = normalize(mul(float4(v.normal, 0.0), unity_WorldToObject).xyz); //Calculate the normal
                o.pos = UnityObjectToClipPos(v.vertex); //And the position
                o.uv = TRANSFORM_TEX(v.uv, _Tex);

                return o;
            }

            fixed4 frag(v2f i) : COLOR
            {
                float3 normalDirection = normalize(i.normal);
                            float3 viewDirection = normalize(_WorldSpaceCameraPos - i.posWorld.xyz);
                        float3 lightDirection = _WorldSpaceLightPos0.xyz - i.posWorld.xyz * _WorldSpaceLightPos0.w;
                float3 halfDirection = normalize(lightDirection + viewDirection );
                            float3 ambientLighting = UNITY_LIGHTMODEL_AMBIENT.rgb * _Color.rgb; //Ambient component


                float roughnessValue = _Roughness;
                    float F0 = _Fresnel;
                    float k = _Metallic;
                float linearRoughness = roughnessValue * roughnessValue;

                float NdotL = max(0.0, dot(normalDirection, lightDirection));
```

```
                    float3 diffuseReflection = _LightColor0.rgb * _Color.rgb * max(0.0,
dot(normalDirection, lightDirection)); //Diffuse component
                float3 specularReflection =  float3(0.0, 0.0, 0.0);

                if(NdotL>0.0){
                   float NdotV = max(0.0, dot(normalDirection, viewDirection));
                   float VdotH = max(0.0, dot(viewDirection, halfDirection));
                   float NdotH = max(0.0, dot(normalDirection, halfDirection));
                   float NH2 = 2.0* NdotH;
                     float g1 = (NH2 * NdotV) / VdotH;
                     float g2 = (NH2 * NdotL) / VdotH;
                     float geoAtt = min(1.0, min(g1, g2));
                        // roughness (or: microfacet distribution function)
                    // beckmann distribution function
                     float r1 = 1.0 / ( 4.0 * linearRoughness * linearRoughness * pow(NdotH,
4.0));
                     float r2 = (NdotH * NdotH - 1.0) / (linearRoughness * linearRoughness *
NdotH * NdotH);
                        float roughness = r1 * exp(r2);
                        // fresnel
                     // Schlick approximation
                     float fresnel = pow(1.0 - VdotH, 5.0);
                     fresnel *= (1.0 - F0);
                     fresnel += F0;


                   //Specular component
                       specularReflection =  _LightColor0.rgb * _SpecColor.rgb * ((fresnel *
geoAtt * roughness) / (NdotV * NdotL * 4));
                }

                 float3 color = (ambientLighting + diffuseReflection) * tex2D(_Tex, i.uv) + (k
+ specularReflection * (1.0 - k)); //Texture is not applient on specularReflection
                return float4(color, 1.0);
             }
          ENDCG
      }
   }
}
```

4.5 Walter (G = Smith, F = Schlick)

```
Shader "Custom/ct_ggx_schlick"
{
    Properties {
      _Color ("Color", Color) = (1, 1, 1, 1) //The color of our object
      _Tex ("Pattern", 2D) = "white" {} //Optional texture

      _Roughness ("Roughness", Range(0.000000001,1)) = 0.5
      _Fresnel("Fresnel Value", Float) = 0.028
      _SpecColor ("Specular Color", Color) = (1, 1, 1, 1) //Specular highlights color
   }
```

```
    SubShader {
        Tags { "RenderType" = "Opaque" } //We're not rendering any transparent objects
        LOD 200 //Level of detail


        Pass {
            Tags { "LightMode" = "ForwardBase" } //For the first light

            CGPROGRAM
                #pragma vertex vert
                #pragma fragment frag
                #pragma target 3.0
                #define PI 3.1415936
                    #include "UnityCG.cginc" //Provides us with light data, camera information,
etc

                uniform float4 _LightColor0; //From UnityCG

                sampler2D _Tex; //Used for texture
                float4 _Tex_ST; //For tiling

                uniform float4 _Color; //Use the above variables in here
                uniform float4 _SpecColor;
                uniform float _Fresnel, _Metallic, _Roughness;

                struct appdata
                {
                    float4 vertex : POSITION;
                    float3 normal : NORMAL;
                    float2 uv : TEXCOORD0;
                };

                struct v2f
                {
                    float4 pos : POSITION;
                    float3 normal : NORMAL;
                    float2 uv : TEXCOORD0;
                    float4 posWorld : TEXCOORD1;
                };

                v2f vert(appdata v)
                {
                    v2f o;

                        o.posWorld = mul(unity_ObjectToWorld, v.vertex); //Calculate the world
position for our point
                     o.normal = normalize(mul(float4(v.normal, 0.0), unity_WorldToObject).xyz);
//Calculate the normal
                    o.pos = UnityObjectToClipPos(v.vertex); //And the position
                    o.uv = TRANSFORM_TEX(v.uv, _Tex);

                    return o;
```

```
            }

            fixed4 frag(v2f i) : COLOR
            {
               float3 normalDirection = normalize(i.normal);
                              float3 viewDirection = normalize(_WorldSpaceCameraPos -
i.posWorld.xyz);
                        float3 lightDirection = _WorldSpaceLightPos0.xyz - i.posWorld.xyz *
_WorldSpaceLightPos0.w;
               float3 halfDirection = normalize(lightDirection + viewDirection );
                           float3 ambientLighting = UNITY_LIGHTMODEL_AMBIENT.rgb *
_Color.rgb; //Ambient component


               float roughnessValue = _Roughness;
               float alpha = roughnessValue * roughnessValue;
               float alphaSqr = alpha * alpha;
                  float F0 = _Fresnel;
                  float k = alpha/2.0f;


               float NdotL = max(0.0, dot(normalDirection, lightDirection));

                        float3 diffuseReflection = _LightColor0.rgb * _Color.rgb * max(0.0,
dot(normalDirection, lightDirection)); //Diffuse component
               float3 specularReflection =  half3(0.0, 0.0, 0.0);

               if(NdotL>0.0f){
                  float NdotV = max(0.0, dot(normalDirection, viewDirection));
                  float LdotH = max(0.0, dot(lightDirection, halfDirection));
                  float NdotH = max(0.0, dot(normalDirection, halfDirection));

                  float F, D, G;

                  float denom = NdotH* NdotH*(alphaSqr - 1.0) + 1.0f;
                  D = alphaSqr /(PI * denom * denom);

                     F = F0 + (1.0-F0)*(pow(1.0 - LdotH, 5.0));

                  float G1 = 1.0f/(NdotV*(1.0f-k)+k);
                  float G2 = 1.0f/(NdotL*(1.0f-k)+k);
                  G = G1 * G2;

                  //Specular component
                        specularReflection =  _LightColor0.rgb * _SpecColor.rgb * ((F * G *
D)/4*NdotL*NdotV);
               }

                float3 color = (ambientLighting + diffuseReflection) * tex2D(_Tex, i.uv) + (k
+ specularReflection * (1.0 - k)); //Texture is not applient on specularReflection
               return float4(color, 1.0);
            }
```

```
            ENDCG
        }
    }
}
```

4.6 Walter (G = Heitz, F = Schlick)

```
Shader "Custom/ct_hcggx_schlick"
{
    Properties {
        _Color ("Color", Color) = (1, 1, 1, 1) //The color of our object
        _Tex ("Pattern", 2D) = "white" {} //Optional texture

        _Roughness ("Roughness", Range(0.000000001,1)) = 0.5
        _Fresnel("Fresnel Value", Float) = 0.028
        _SpecColor ("Specular Color", Color) = (1, 1, 1, 1) //Specular highlights color
    }
    SubShader {
        Tags { "RenderType" = "Opaque" } //We're not rendering any transparent objects
        LOD 200 //Level of detail


        Pass {
            Tags { "LightMode" = "ForwardBase" } //For the first light

            CGPROGRAM
                #pragma vertex vert
                #pragma fragment frag
                #pragma target 3.0
                #define PI 3.1415936
                    #include "UnityCG.cginc" //Provides us with light data, camera information,
etc

                uniform float4 _LightColor0; //From UnityCG

                sampler2D _Tex; //Used for texture
                float4 _Tex_ST; //For tiling

                uniform float4 _Color; //Use the above variables in here
                uniform float4 _SpecColor;
                uniform float _Fresnel, _Metallic, _Roughness;

                struct appdata
                {
                    float4 vertex : POSITION;
                    float3 normal : NORMAL;
                    float2 uv : TEXCOORD0;
                };

                struct v2f
                {
                    float4 pos : POSITION;
                    float3 normal : NORMAL;
```

```
            float2 uv : TEXCOORD0;
            float4 posWorld : TEXCOORD1;
        };

        v2f vert(appdata v)
        {
            v2f o;

                o.posWorld = mul(unity_ObjectToWorld, v.vertex); //Calculate the world position for our point
                o.normal = normalize(mul(float4(v.normal, 0.0), unity_WorldToObject).xyz); //Calculate the normal
            o.pos = UnityObjectToClipPos(v.vertex); //And the position
            o.uv = TRANSFORM_TEX(v.uv, _Tex);

            return o;
        }

        fixed4 frag(v2f i) : COLOR
        {
            float3 normalDirection = normalize(i.normal);
                        float3 viewDirection = normalize(_WorldSpaceCameraPos - i.posWorld.xyz);
                    float3 lightDirection = _WorldSpaceLightPos0.xyz - i.posWorld.xyz * _WorldSpaceLightPos0.w;
            float3 halfDirection = normalize(lightDirection + viewDirection );
                    float3 ambientLighting = UNITY_LIGHTMODEL_AMBIENT.rgb * _Color.rgb; //Ambient component


            float roughnessValue = _Roughness;
            float alpha = roughnessValue * roughnessValue;
            float alphaSqr = alpha * alpha;
                float F0 = _Fresnel;
                float k = alpha/2.0f;

            float NdotL = max(0.0, dot(normalDirection, lightDirection));

                    float3 diffuseReflection = _LightColor0.rgb * _Color.rgb * max(0.0, dot(normalDirection, lightDirection)); //Diffuse component
            float3 specularReflection =  half3(0.0, 0.0, 0.0);

            if(NdotL>0.0){
                float NdotV = max(0.0, dot(normalDirection, viewDirection));
                float LdotH = max(0.0, dot(lightDirection, halfDirection));
                float NdotH = max(0.0, dot(normalDirection, halfDirection));

                float F, D, G;

                float denom = NdotH* NdotH*(alphaSqr - 1.0) + 1.0f;
                D = alphaSqr /(PI * denom * denom);
```

```
                    F = F0 + (1.0-F0)*(pow(1.0 - LdotH, 5.0));

                float GGXV = NdotL* sqrt(NdotV * NdotV * (1.0 - alphaSqr) + alphaSqr);
                float GGXL = NdotV * sqrt(NdotL * NdotL * (1.0 - alphaSqr) + alphaSqr);
                G = 0.5f/(GGXV + GGXL);

                //Specular component
                specularReflection =  _LightColor0.rgb * _SpecColor.rgb * ((F * G * D));
            }

             float3 color = (ambientLighting + diffuseReflection) * tex2D(_Tex, i.uv) + (k
+ specularReflection * (1.0 - k)); //Texture is not applient on specularReflection
            return float4(color, 1.0);
        }
      ENDCG
    }
  }
}
```

4.5 Oren-Nayar shader [27]

```
Shader "Custom/oren-nayar"
{
 Properties {
  _Albedo ("Albedo", Color) = (1, 1, 1, 1)
  _Roughness ("Roughness", Range(0, 1)) = 0.5
  _Tex ("Pattern", 2D) = "white" {}
 }
 SubShader {
  Tags { "RenderType" = "Opaque" }
  LOD 200

  Pass {
   Tags { "LightMode" = "ForwardBase" }

   CGPROGRAM
   #pragma vertex vert
   #pragma fragment frag
   #pragma target 3.0
   #include "UnityCG.cginc"
   // Properties
   uniform fixed4 _LightColor0;
   uniform fixed4 _Albedo;
   uniform half  _Roughness;
   sampler2D _Tex; //Used for texture
   float4 _Tex_ST; //For tiling

   // Vertex Input
   struct appdata {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float2 uv : TEXCOORD0;
   };
```

```
    struct v2f {
      float4 pos      : SV_POSITION;
      float3 normal   : NORMAL;
      float2 uv : TEXCOORD0;
      float4 posWorld : TEXCOORD1;
    };

    v2f vert(appdata v) {
      v2f o;
      o.pos      = UnityObjectToClipPos(v.vertex);
      o.normal   = normalize(mul(v.normal, unity_WorldToObject).xyz);
      o.uv = TRANSFORM_TEX(v.uv, _Tex);
      o.posWorld = mul(unity_ObjectToWorld, v.vertex);
      return o;
    }

    fixed4 frag(v2f i) : SV_Target {
      // Vector
      half3 normal   = normalize(i.normal);
      half3 lightDir = normalize(_WorldSpaceLightPos0.xyz);
      half3 viewDir  = normalize(_WorldSpaceCameraPos.xyz - i.posWorld);

      // Dot
      half NdotL = saturate(dot(normal, lightDir));
      half NdotV = saturate(dot(normal, viewDir));

      // Roughness
      half roughness    = _Roughness * _Roughness;
      half roughnessSqr = roughness * roughness;

      // Oren-Nayar
      half A = 1.0 - 0.5 * roughnessSqr / (roughnessSqr + 0.33);
      half B = 0.45 * roughnessSqr / (roughnessSqr + 0.09);
         half C = saturate(dot(normalize(viewDir - normal * NdotV), normalize(lightDir -
normal * NdotL)));
      half angleL = acos(NdotL);
      half angleV = acos(NdotV);
      half alpha  = max(angleL, angleV);
      half beta   = min(angleL, angleV);
            fixed3 diffuse = _Albedo.rgb * (A + B * C * sin(alpha) * tan(beta)) *
_LightColor0.rgb * NdotL;

      // Color
      fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.rgb * _Albedo.rgb;
      fixed4 color = fixed4(ambient + diffuse, 1.0);

      return color*tex2D(_Tex, i.uv);
    }
    ENDCG
  }
 }
```

```
}
```

4.6 Disney shader

```
Shader "Custom/disney"
{
  Properties {
    _Albedo ("Albedo", Color) = (1, 1, 1, 1)
    _Roughness ("Roughness", Range(0, 1)) = 0.5
    _Tex ("Pattern", 2D) = "white" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    LOD 200

    Pass {
      Tags { "LightMode" = "ForwardBase" }

      CGPROGRAM
      #pragma vertex vert
      #pragma fragment frag
      #pragma target 3.0
      #include "UnityCG.cginc"
      #define PI 3.1415926
      // Properties
      uniform fixed4 _LightColor0;
      uniform fixed4 _Albedo;
      uniform half   _Roughness;
      sampler2D _Tex; //Used for texture
      float4 _Tex_ST; //For tiling

      // Vertex Input
      struct appdata {
        float4 vertex : POSITION;
        float3 normal : NORMAL;
        float2 uv : TEXCOORD0;
      };

      // Vertex to Fragment
      struct v2f {
        float4 pos      : SV_POSITION;
        float3 normal   : NORMAL;
        float2 uv : TEXCOORD0;
        float4 posWorld : TEXCOORD1;
      };

      //-----------------------------------------------------------------
      // Vertex Shader
      //-----------------------------------------------------------------
      v2f vert(appdata v) {
        v2f o;
```

```
        o.pos      = UnityObjectToClipPos(v.vertex);
        o.normal   = normalize(mul(v.normal, unity_WorldToObject).xyz);
        o.uv = TRANSFORM_TEX(v.uv, _Tex);
        o.posWorld = mul(unity_ObjectToWorld, v.vertex);
        return o;
      }

      //-----------------------------------------------------------------
      // Fragment Shader
      //-----------------------------------------------------------------
      fixed4 frag(v2f i) : SV_Target {
        // Vector
        half3 normal   = normalize(i.normal);
        half3 lightDir = normalize(_WorldSpaceLightPos0.xyz);
        half3 viewDir  = normalize(_WorldSpaceCameraPos.xyz - i.posWorld);
        float3 halfDir = normalize(lightDir + viewDir );
        // Dot
        half LdotH = saturate(dot(lightDir, halfDir));
        half NdotL = saturate(dot(normal, lightDir));
        half NdotV = saturate(dot(normal, viewDir));

        // Roughness
        half roughness    = _Roughness;


        half f90 = 0.5 + 2.0 * roughness * LdotH * LdotH;
        half lightScatter = 1.0 + (f90 - 1.0)* pow(1.0 - NdotL,5.0);
        half viewScatter = 1.0 + (f90 - 1.0)* pow(1.0 - NdotV,5.0);

        fixed3 diffuse =  _Albedo.rgb *  _LightColor0.rgb * lightScatter * viewScatter * (1.0 /
PI)  ;
        // Color
        fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.rgb * _Albedo.rgb;
        fixed4 color = fixed4(ambient + diffuse, 1.0);

        return color*tex2D(_Tex, i.uv);
      }
      ENDCG
    }
  }
}
```

REFERENCES

1. Ashikmin, M., Premože, S. and Shirley, P., 2000, July. A microfacet-based BRDF generator. In Proceedings of the 27th annual conference on Computer graphics and interactive techniques (pp. 65-74).
2. Blinn, James F. "Models of light reflection for computer synthesized pictures." Proceedings of the 4th annual conference on Computer graphics and interactive techniques. 1977.
3. Burley, B. and Studios, W.D.A., 2012, August. Physically-based shading at disney. In ACM SIGGRAPH (Vol. 2012, pp. 1-7). vol. 2012.
4. Beckmann, P. and Spizzichino, A., 1987. The scattering of electromagnetic waves from rough surfaces. Norwood.
5. Clarke, F. J. J., and D. J. Parry. "Helmholtz reciprocity: Its validity and application to reflectometry." Lighting Research & Technology 17.1 (1985): 1-11.
6. Cook, R.L. and Torrance, K.E., 1982. A reflectance model for computer graphics. ACM Transactions on Graphics (ToG), 1(1), pp.7-24.
7. Guarnera, D., Guarnera, G. C., Ghosh, A., Denk, C., & Glencross, M. (2016, May). BRDF representation and acquisition. In Computer Graphics Forum (Vol. 35, No. 2, pp. 625-650).
8. Heitz, E., 2014. Understanding the masking-shadowing function in microfacet-based BRDFs. Journal of Computer Graphics Techniques, 3(2), pp.32-91.
9. Hoffman, Naty. "Crafting physically motivated shading models for game development." part of “Physically Based Shading Models in Film and Game Production,” SIGGRAPH (2010).
10. Jensen, Henrik Wann, et al. "A practical model for subsurface light transport." Proceedings of the 28th annual conference on Computer graphics and interactive techniques. 2001.
11. . H. Lambert. Photometria sive de mensura et gradibus luminis, colorum et umbrae (in Latin), 1760.
12. Nicodemus, F. E., & FE, N. (1977). Geometrical considerations and nomenclature for reflectance.
13. Oren, M. and Nayar, S.K., 1994, July. Generalization of Lambert's reflectance model. In Proceedings of the 21st annual conference on Computer graphics and interactive techniques (pp. 239-246).
14. Phong, B.T., 1975. Illumination for computer generated pictures. Communications of the ACM, 18(6), pp.311-317.
15. Christophe Schlick. 1994. An Inexpensive BRDF Model for Physically-Based Rendering. Computer Graphics Forum, 13 (3), 233–246.
16. Smith, B., 1967. Geometrical shadowing of a random rough surface. IEEE transactions on antennas and propagation, 15(5), pp.668-671.
17. Torrance, Kenneth E., and Ephraim M. Sparrow. "Theory for off-specular reflection from roughened surfaces." Josa 57.9 (1967): 1105-1114.
18. Walter, B., Marschner, S.R., Li, H. and Torrance, K.E., 2007. Microfacet Models for Refraction through Rough Surfaces. Rendering techniques, 2007, p.18th.
19. CGRU0 Project. Available: http://cgru.sourceforge.net/src/shaders_p/shaders_p_3.2/docs/p_megatk.html
20. Joey de Vries. Learn OpenGL – Graphics Programming. Available: https://learnopengl.com/book/book_pdf.pdf
21. Montes, Rosana, and Carlos Ureña. "An overview of BRDF models." University of Grenada, Technical Report. LSI-2012-001 (2012).
22. Physical Based Rendering in Filament. Available: https://google.github.io/filament/Filament.html#overview/physicallybasedrendering
23. Kevin George. Shading Diffuse Models. Available: https://kevin-george-2n3x.squarespace.com/blog/2014/5/25/shading-diffuse-models
24. Morten Mikkelsen. GGX in Unity 5.3. Available: https://blog.unity.com/technology/ggx-in-unity-5-3
25. Naty Hoggman. Recent Advances in Physically Based Shading. Available: https://blog.selfshadow.com/publications/s2016-shading-course/hoffman/s2016_pbs_recent_advances_v2.pdf
26. Brian Karis. Real Shading in Unreal Engine 4. Available: https://de45xmedrsdbp.cloudfront.net/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf
27. BASICxSHADER. Available: https://github.com/midnightSuyama/BASICxSHADER/blob/master/shaders/Lighting/Oren-Nayar.shader

28. Shaders in Unity. Available: https://medium.com/@deshankalupahana/shaders-in-unity-lambert-and-ambient-ceba9fab6cfa
29. Cook-Torrance Shader. Available: https://gist.github.com/jose-villegas/3f15eb72fb4a818e0ba5
30. Space Robot Kyle Asset. Available: https://assetstore.unity.com/packages/3d/characters/robots/space-robot-kyle-4696
31. Lambertian Surface. Available: https://birdingimagequalitytool.blogspot.com/2015/06/forensics-lamberts-cosine-law.html
32. Fresnel effect on water. Available: https://www.reddit.com/r/itookapicture/comments/hs9uw0/itap_of_a_perfection_reflection_at_lake_tahoe/
33. Hammon, E., L. S. Engineer, and R. Entertainment. "Pbr diffuse lighting for ggx+ smith microsurfaces." *Game Dev. Conf*. 2017.
34. Phong Shader. Available: https://janhalozan.com/2017/08/12/phong-shader/