Real-time Graphical Techniques

https://youtu.be/8PBi2Nxfmbo

Jainesh Pathak
August 2023
School of Computing Science
Newcastle University
J.P.Pathak2@newcastle.ac.uk

ABSTRACT

Lighting physics in the real world is complex and too much expensive for the processor to handle such computations. The department of computer graphics has undergone numerous transformations and revolutionized the way computer generated images are produced and perceived from drawing 2D lines to drawing complex 3D geometry with the help of fast parallel processing of the GPU.

One such transformative shift is the introduction and widespread adoption of Physically Based Rendering (PBR) which tries to bridge the gap between virtual and physical reality, and which is basically the approximation of the real-world lighting physics in the computer world. PBR's core components works on the utilization of microfacet theory, Fresnel equations and a special function called BRDF and doesn't violate the law of energy conservation. For PBR to work at perfectly, the image needs to gamma corrected. While generating images from different light sources is good, it can be further augmented using Image-based Lighting or IBL which takes the light information from a single texture.

Although, PBR is part of lighting calculations which generates beautiful images. Those images can be further enhanced using the term called Post Processing. There are quite many post processing effects whose functions differ from other effects like Bloom, Vignette, Ambient Occlusion, Motion Blur, Auto Exposure, Depth of Field, etc. Post Processing is heavily used in static images video games and visual effects of movies.

This report serves as a demonstration of PBR explaining the algorithms in-depth, investigating factors like roughness, material surface properties, view direction, and few Post Processing effects like Bloom, SSAO, Vignette. Lastly, it will also do comparison of PBR with other well-known shading effects used in the industry like Disney's Principled BSDF, Oren-Nayar and Phong shading.

KEYWORDS

PBR; BRDF; Lighting; Irradiance; Diffuse; Specular; Post-Process

1 INTRODUCTION

In computer graphics, lighting plays a vital role of making the player/viewer believe the realism of the digital worlds. In old arcade style games like Pac-Man, Wolfenstein 3D, lighting was non-existent, and player can see the whole level or map even at huge distance (E.g., A corridor/room in Wolfenstein 3D). Even if expensive lighting algorithms existed, the CPUs of that era were not powerful enough to handle those lighting calculations in real time. Later, as CPUs and GPUs became more powerful, the simulation of realistic lighting also became possible by exploiting the parallel processing power of the GPUs.

The foundations of PBR needs to be properly explained before diving straight into the rendering equations. Shirley [1] explains those basic foundations by first stating that PBR normally has three distinct goals:

- Predictive Rendering: The computer image result matches the scene which is used in design and simulation applications.
- Plausible Rendering: The image result is close to real world but can be bit incorrect and is used in video games and movies.
- Visually Rich Rendering: Visual richness of reality can be stylized.

For this report, the focus is on video games. So, Plausible Rendering is sufficient.

1.1 Light

Lights are called Photon. In real-life, a camera has lens and sensors. The sensor has minute elements called pixels which respond differently depending on the direction and wavelengths of photon. However, in rendering to measure to real light properties like its wavelength both single or multiple and mass is not possible. Wavelengths between 390nm and 700nm are part of visible light spectrum that the human eye can see. For this report, it is assumed that a photon contains some energy, wavelength, time, position and direction.

Directly taking these different wavelengths measures in graphics is not practical. Hence, graphic programmers make use of light colour encoded in the RGB component (Red, Green, Blue) in the range of 0 to 1 for simulating the light. Figure 1 shows the different wavelength of daylight.

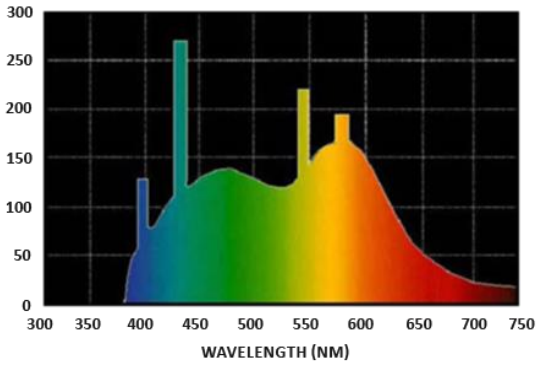

Figure 1: Wavelength of a daylight

1.1.1 Light Sources:

Different light sources were introduced like Point Light, Directional Light and Spotlight.

To quickly summarize these different light sources:

1.1.1.1 Point Light:

This type of light source emits lights in all directions. The intensity gradually attenuates as the distance from surface and light increases. In technical details it contains the position in 3D world, light colour, intensity value and range.

1.1.1.2 Directional Light:

This light emits in the single direction, and it has no attenuation and light travel infinitely in the game world. Just like a sun, however the sun in real life is one big point light in the solar system.

1.1.1.3 Spotlight:

This type of light emits light just like directional light, and it falls in a certain radius. Surfaces outside the radius are not lit at all. A good example of spotlight is flashlight which are common to use in horror games.

1.2 Radiance

Radiance is used to measure the energy emitted, reflected or received by the surface of the object when looking at different angles. Shirley [1] explains that radiance is discussed across many literatures and has come up with few properties of radiance:

- Radiance travel in constant straight line.
- Radiance can be colours that is seen from the direction.
- Camera sensors measure the moments of radiance.
- All light measurements can be derived from radiance.

Number 1 and 2 are similar in the sense that photons never lose any energy as they travel, but this easily violates the light attenuation law.

Therefore, radiance can be defined as a function over space:

Radiance = L(position, direction, wavelength, time)

Where L is the Spectral Radiance. Spectral Radiance is the emitting radiance spread across different wavelengths. In other words, it tells us how much light energy is transmitted off the surface at which wavelengths.

This function is adopted almost everywhere except for surfaces where reflection/refraction happens. As a result, there are two kinds of radiance:

- Incoming Radiance: Incoming direction of the light of what the surface sees.
- Outgoing Radiance: Direction that the camera/eye sees when looking at the surface.

1.3 Irradiance

The continuous sum of all radiance is called Irradiance at point x. The integral or constant function is denoted as H which is the density of light coming from all directions landing on a surface.

The irradiance equation for incoming radiance is given as:

$$H(x,\lambda) = \int_{\omega\ in} L(x,\omega,\lambda)\cos\theta\ \mathrm{d}\omega$$

Here, ω is the incoming light direction and θ is the angle or dot product between surface normal and light direction.

The integral function is the same for outgoing radiance:

$$H(x,\lambda) = \int_{\omega\ out} L(x,\omega,\lambda)\cos\theta\ \mathrm{d}\omega$$

1.4 Lambertian Surface

A *matte* is a surface that is not reflective or shiny and most of the surfaces are *matte*. They are usually common in computer graphics and are made with the help of *Lambertian Surface* principles.

The basic property of a *Lambertian Surface* is that the brightness of a surface to the camera/eye is same regardless of the angle of view. The surface normal are isotropic or evenly distributed.

The Lambertian diffuse reflectance is the most used in computer graphics due its easy formula:

$$L(\lambda) = \frac{1}{\pi}E = \frac{C(\lambda)}{\pi}H$$

C is the albedo or flat colour of the surface. Here, π is used to satisfy the energy conservation law which means the surface should emit same amount of energy it receives.

Once, the Lambertian result is achieved, it is finally multiplied with the irradiance equation *H*.

1.5 Rendering Equation

The rendering equation especially for computer graphics was first introduced by Kajiya in 1986. It tells about the radiance of the surface to the incoming radiances. The general rendering equation is given as:

$$L(\omega o) = Le(\omega o) + \int_{\omega i} p(\omega i, \omega o) L(\omega i) cos\theta d\omega i$$

In summary, the rendering equation tells that light that comes in for all directions is either reflected or refracted. Here, p is the Bidirectional Reflectance Distribution Function (BRDF) and it depends on the surface properties like roughness. Also, *i* and *o* are the incoming and outgoing radiances and *Le* is the emitted part.

θ is the angle or dot product between surface normal and light direction. This equation is further explained in detail later in the report.

1.6 BRDF

A Bidirectional Reflectance Distribution Function (BRDF) is a function that shows how a light radiance is reflected from an opaque surface. It takes inputs ωi and ωo as incoming and outgoing light and surface normal n. It returns ratio of light direction along the viewing direction and irradiance on the surface from the incoming light direction.

BRDFs comes in two types:

- Diffuse BRDF (Refraction)
- Specular BRDF (Reflection)

The diffuse BRDF takes care of how incoming light energy is absorbed or refracted into the surface. The most famous diffuse BRDF is the lambert which is explained above in section 1.4. Lambert diffuse is famous for its simplicity but produces incorrect results and doesn't work on micro surface level and it is not the only BRDF. There are other diffuse BRDF like the Oren-Nayar and Burley BRDF which is implemented and will be explained later in this report.

The second type is called the Specular BRDF which shows how the light is reflected depending on the roughness property of the surface. It requires more computations than the diffuse BRDFs. There are several variants specular functions like the Cook-Torrance, Beckmann, GGX, Gaussian, etc. Some variants are accompanied with Fresnel effects. The previous mentioned BRDFs are also implemented and is further explained in the report. Figure 2 shows light reflections in gray colour depending on the type of surface.

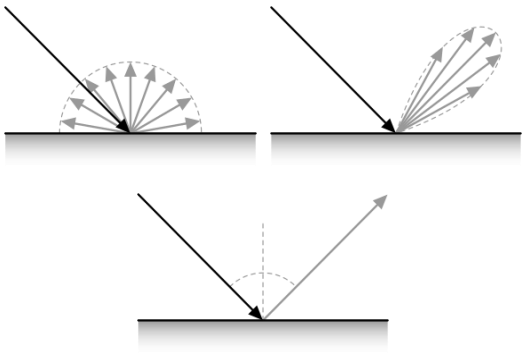

Figure 2: Clockwise from top left: Diffuse, Glossy, Mirror

2 BACKGROUND AND RELATED WORKS

2.1 Lighting in Doom

One of the greatest games of all time – Doom and others like Heretic, Hexen did show some sort of dynamic lighting although not realistic which is called Sector-based lighting.

In Doom game, the world is divided into sectors which are made by connecting line segments to form a polygon. Every sector had different properties like floor, ceiling, texture and most importantly the light levels. Every sector had a light level ranging from 0-255 with 0 being dark and 255 being very bright.

Lighting attenuation computations were done such that the light travel from one sector to its adjacent sectors and based on the distance of how much it traveled, the light level of current sector will be slowly attenuated. It did support additional light sources like lamp for doing light computation for that sector and its adjacent ones.

2.1.1 Light Diminishing:

Another fine addition to lighting Doom introduced is Light Diminishing or Distance Fading. Where the brightness of the area from the player's point of view slowly decreases as the distance from the player increases. This did not create any realism as well, but it did create impressive scary atmosphere for the game. This same mechanism was also used to simulate fogs [1].

Figure 3 shows the distance fading effect. Bandings can be clearly seen on walls and floor. The reason for banding that can be clearly seen on floor and wall is explained further in colourmap implementation.

Figure 3: Light Diminishing. Banding effect is seen on floor and walls.

2.1.2 Colourmap implementation:

Before explaining Colourmap, I need to first explain what colour quantization is.

2.1.2.1. Colour quantization

Quantization in general is an image compression technique which is used to narrow down certain range of values to a single value. This also helps in reducing the file size. Moreover, colour quantization works by reducing colours of the image such that the compressed is visually like its original.

In the PC port of original doom engine, it made use of a Colourmap which is a colour quantized gradient texture which starts from a different colour starting at top and slowly fades to black colour as moving down. It is a precomputed lookup table which was used to fit in the game's 256-colour palette. The first 32 levels in the colourmap were purely dedicated for implementing lighting. See Figure 4 for example.

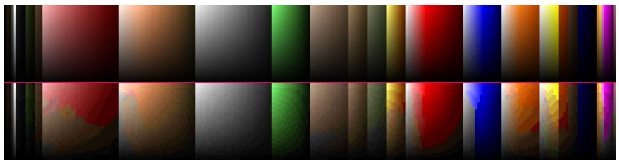

Figure 4: Top – Pure gradient. Bottom – Doom's quantized colourmap light levels.

2.1.3 Fake Contrast:

Fake contrast is a feature in Doom engine where the walls parallel to east-west axis are made darker compared to walls facing north-south. Although, this feature dated back in older games like Hovertank 3D. However, this was not perfect because it works only on orthogonal geometry.

2.1.4 Coloured Lighting:

Coloured Lighting is another feature in Doom engine where the ambient lighting in a sector can be multiplied with a tint colour. This feature was heavily used in PlayStation port of Doom.

Doom 64 had more sophisticated coloured lighting where every sector has a maximum support of five different coloured lights and each of them affect only one type of entity like sprites, floor, ceiling, etc.

2.1.4.1 PlayStation Special Lighting:

The PlayStation port of Doom had a feature of special lighting effects that can affect only walls or floors. It makes use of a table of 256 32-bit binary colour values which are stored in RGBA order. Every sector in PS1's map format has a single byte property which stores the index pointing to the colour table for determining the special lighting effect. Figure 5 shows the light colour information.

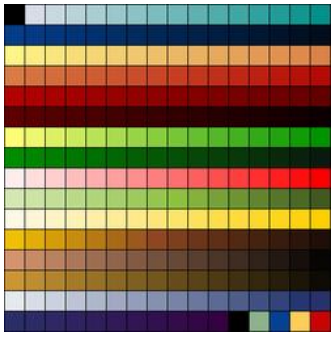

Figure 5: PS1's Special Lighting colour table.

In conclusion, the lighting calculations in Doom was simple with only support of ambient lighting and with some special lighting effects did contribute to bring immersive gameplay and visually impressive atmosphere.

Later, as the engine was released as open-source, several third-party or fan-made projects were released with enhances and integration of modern graphics API like OpenGL, Vulkan with dynamic lightings.

2.2 Shading Models

In computer graphics, the most popular shading model for specular highlights in the Phong shading model which he has explained in detail [2]. His work is influenced from previous shading model such that of Warnock's [3] and Newell, Newell, Sancha's shading model [4] and Gouraud [5] shading model which needs to be explained first.

2.2.1 Warnock's Shading:

Back in early computer graphics time, the most common approach for rendering polygons is the Scanline Rendering approach which works on horizontal line-by-line basis. The main objective of this rendering is to provide depth perception which will be draw on 2D screen. For each scanline, it determines the visible parts of polygons and computers the shading for those visible areas. The polygons that are to be rendered are first sorted in Y coordinates and later by scanline checks for any intersection of a scanline to determine which polygon is on front and is added to the sorted list. This sorted list is updated as the scanline moves down the screen. The benefits on this sorting algorithm were that there was no need to do any edges comparison on per-pixel level.

Shading algorithm developed by Warnock [3] works on scanline rendering approach. His algorithm is also often called as Flat-Shading and is one of the foundations for modern shading algorithms.

His algorithm works by putting light source and camera are placed at the same position and function returns the sum of two terms used: Surface Normal and Specular reflection. The reason for putting the light source and camera at same position is to avoid the shadow problem. Surfaces facing towards the light source are always bright compared to its adjacent which were shaded with different light intensities. The polygon surfaces that move away from the light source receive less light intensity and hence appear darker. Also, discontinuity can be easily seen on curved surfaces as seen in Figure 6.

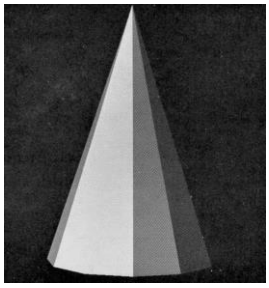

Figure 6: Warnock's Shading destroys smoothness on curved surfaces.

His shading algorithm uses the rule:

$$S = |\frac{cos\ \theta}{R}|$$

Where θ is the angle between light direction and surface normal and R is the measure of distance from the observer and returns the absolute value.

The advantage of this algorithm is that it was efficient enough to draw complex scenes on low-end hardware in less time and provided visually believable rendering for flat surfaces. Unfortunately, his shading assumes that the 3D models are always convex and hence doesn't work on concave polygons.

2.2.2 Newell, Newell, Sancha's Shading:

While Warnock's and other shading works well for opaque and flat-shaded model, they don't work for cases where object contains transparency.

The Newell-Sancha [4] model is an extension of previous shading models that does the depth sorting in a different manner and handles the shading of transparent objects as well.

Their depth sorting algorithm works by first having a linear array of data items which contains plane equations in screen-space and values of x and y on screen. Additionally, it also has an ordered list which stores the references of faces in order of minimum z values where that minimum z tells the point furthest from the camera. The top of the list is then compared with other points to check whether that other point is drawing over the current face. If the current face is not obscuring, then it is removed from top of the list and added to the screen map used for rendering and moves to the next point and does the same comparison.

However, there were also edge-cases where the two faces fail to determine there are obscuring or not. In such case, one of the faces is split into two faces and check if the second face lies on both recently split faces of current face. If it does, then second face is also split and is added in the ordered list for depth comparison.

For performing light shading operations, they followed same method as Warnock's by keeping the light and camera at same position with the reason to avoid shadow problem. Object faces were shaded using different light intensities using the existing simple function:

$$I = r.\cos^n\ (a) + b$$

Here, *r* is the light intensity range, n being some random power, *a* is angle between the incoming light and face normal and *b* is the ambient light level. With *n* being 1, it simulated diffuse reflection, but as power is increased for example 20, then it leads to objects appear much darker that are barely facing the light and those facing appears the brightest.

Transparent faces are simulated by having the line segments added to the screen map instead of overwriting it and using a different intensity function:

$$I1 < I0 : I = w.I1 + (1 - w).I0$$
$$I1 > I0 : I = I1$$

Where I1 and I0 are the intensities of newly created line segment and the one that is overdrawn, and *w* is the weight factor. Figure 7 shows the shading effect of transparent object.

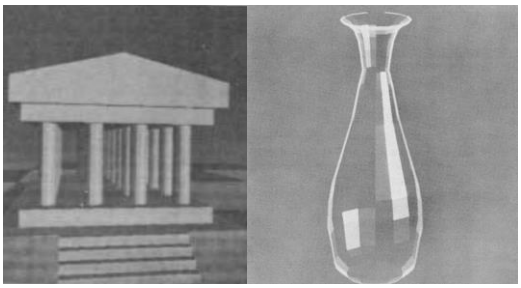

Figure 7: Newell, Newell, Sancha's Shading.

The intensity functions for transparent faces were more of a hack than simulating real world lighting on non-opaque objects. But they were efficient enough to run on limited hardware.

Overall, the shading of Newell-Sancha model was still flat-shaded with a fine addition of handling transparent surfaces. They were still not good enough for curved surfaces and the problem of Mach banding effect still occurs. The Mach banding effect problem is explained further next section.

2.2.3 Gouraud Shading:

Warnock and Newell-Sancha shading were visually impressive but were prone to problem of Mach bands effects when it comes to curved surfaces.

2.2.3.1 Mach Band Effect:

Mach bands [8] is a psychological illusion that triggers the edge detection in the human visual system. It is dependent on the intensity of light. Every neuron that it receives light information communicates with its neighbour and modifies the performance. It visualizes discontinuities as seen in Figure 3, 6, 7. In other words, the banding effect occurs when there is a jump in contrast values between two surfaces.

2.2.3.2 Curved Surfaces:

Coons [7] introduced the rendering the polygons with curved surfaces. It also known as Coons patch. The rendering of polygons using the Coons patch were grid-based or wireframe and didn't have any support of depth sorting and as a result, back surfaces were also rendered. Several variants were introduced using Coons patch as a base layer. Figure 8 shows an example of Coons patch curved surface.

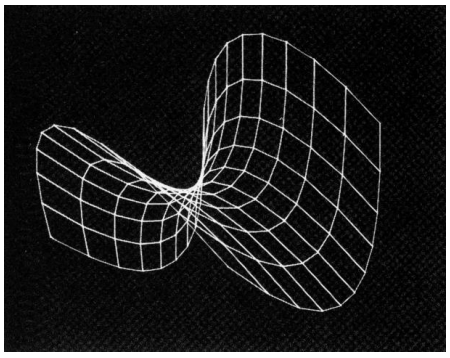

Figure 8: Coons Patch model of hyperbole.

Coons patch didn't have any feature of light shading and running Warnock's algorithm easily produces Mach bands as it can be seen in Figure 9.

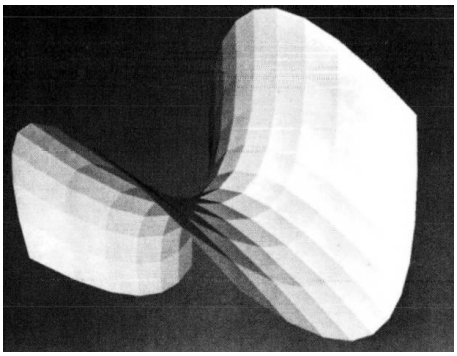

Figure 9: Warnock's Shading on Coons Patch.

Gouraud [5] developed the shading algorithm which computes the specular highlights on curved surfaces of each vertex of the triangle. With the rule of shading that it needs to continuous for polygons and its neighbours. For achieving that continuity, every vertex has one normal which is computed as mean of all normal of each polygon associated with that vertex.

For example, if a polygon has vertices A, B, C, D and projection edges are AB and CD and a scanline (Figure 10) with E, F being the intersection point of scan line with edges AB and CD and P being any arbitrary point on the scanline, then the shading value at point P can be calculated as linear interpolation of SA and SB in formula:

$$Se = (1 - a) * SA + a * SB$$

Here, a is the coefficient ($0 \leq a \leq 1$) showing the position of E on the edge AB. The same formula can be calculated for SF and SP.

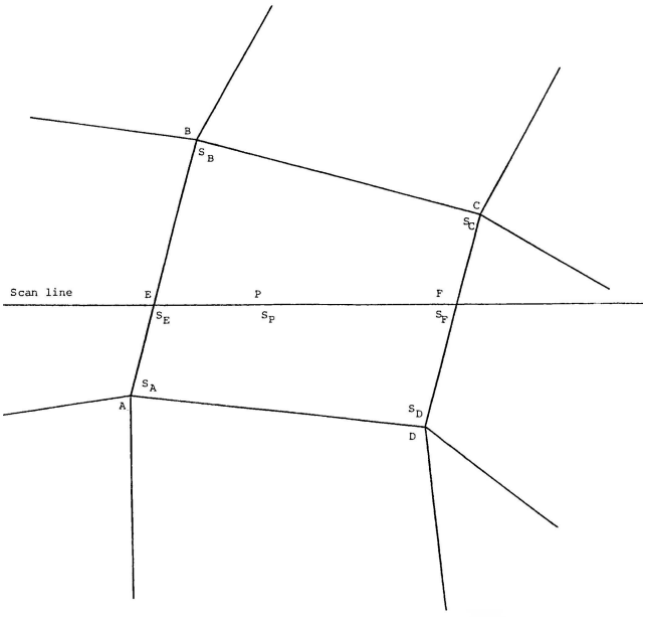

Figure 10: Projection of a polygon with scanline.

This algorithm was demonstrated using the same scanline rendering approach. This generated visually appealing shading effect and retained the smoothness of curved surfaces as it can be seen in Figure 11.

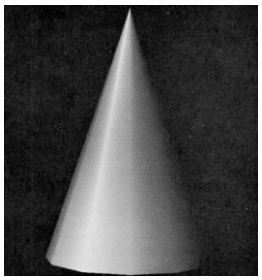

Figure 11: Gouraud Shading effect on low polygon cone.

Unfortunately, the algorithm was still not perfect, and it still created same discontinuity issue if the object has low polygons. If

the surface of the object contains high amount of specular reflection, then the highlights are often irregularly shaped because it depends on the shape of the polygons approximate curved surface and not the curved surface of object. In computer generated films, the frame-by-frame discontinuities can be seen even more when the object is in motion, in one frame the faces which are orthogonal to direction of light ray takes a uniform shade and in the next frame, those same faces are in different orientation towards the light source. This same effect happens, when the object's orientation is changed.

This problem can be solved if the number of polygons is increased, but this is not an efficient solution, because of high memory amount required. Here's where enters the Phong shading model.

2.2.4 Phong Shading:

Bui Tuong Phong [2] introduced new shading model heavily inspired from Gouraud shading. It consists of three core components viz. Ambient, Diffuse and Specular.

Ambient lighting is where Object is never dark, there is still some amount light in the world. It is a simple constant value that is added to object's surface colour. In simple term, Ambient light is omni-present even when there is not a single light source.

Diffuse lighting is an important which directly simulates the light ray on the object's surface. Surfaces that face the light source are brighter than the ones that are not facing.

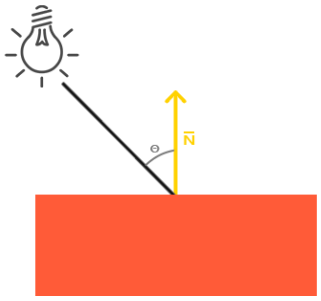

Figure 12: Diffuse Light Ray and surface normal N used to calculate the cosine angle.

It is calculated by taking an incident ray or light ray direction vector, the normal vector which perpendicular to the object's surface and thereby calculating the dot product or the cosine angle of these two vectors. If the angle is greater than zero means the surface of object is facing the light source and vice versa. That is why it is best to check if dot product is always greater than zero beforehand.

To calculate light ray direction is simple by subtracting the light's position and surface position and normalizing it.

The normal vector is calculated from a triangle of a mesh. It is stored with the mesh data file as it is not required to calculate it every frame. To calculate the normal vector, we first take a difference vector from the two vertices to the one vertex and normalizing it and taking the cross product of those difference vectors gives the normal vector. However, the order of cross products matter.

Once the angle is retrieved, it is simply multiplied with the light colour and lastly added with the ambient component to create a believable image. This can be seen in Figure 13 with ambient lighting.

Figure 13: Diffuse + Ambient Lighting on the mesh.

Specular lighting computes the bright highlighted spot on the surface of the object depending on the shininess value of the surface.

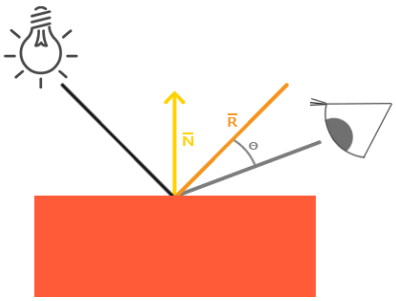

Figure 14: Specular lighting introduces View direction vector.

It takes same input from diffuse i.e., the light direction which is a normalized difference vector of light position and surface position in world space and a surface normal. Additionally, it takes the third input which is a direction vector which is obtained by normalized difference of camera/eye position and surface position again in world space. Then the reflection vector is calculated from the light ray direction and surface normal.

Lastly the dot product or cosine angle is done from the view direction and reflection vector and is again made sure that its value is positive. This dot product is then raised by the shininess value constant. A high shininess value tightens the bright spot. See Figure 16. The reflection vector can be calculated using the formula:

$$R = 2(N \,.\, L)N - L$$

Here, *R* is the reflection vector and *N* is the surface normal and light is incident light direction. Figure 15 shows the breakdown example.

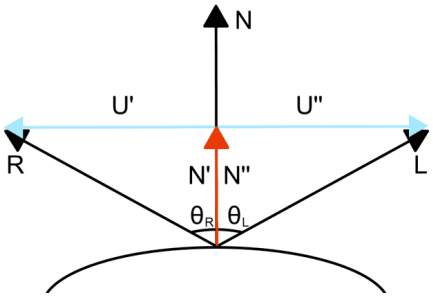

Figure 15: Reflection Vector

Based on the law reflection:

$$\theta R = \theta L$$

θR is the dot product or cosine of angle between vector *R* and normal *N'* and same for vector *L* and normal *N''*. The above formula can be rewritten as:

$$R.N = L.N$$

From figure 12, it is also known that vector *U'* = -*U''*. Since, *U'* is difference between *R* and *N'*, so the reflection can be written as:

$$U' = R - N' = R - (R.N')N'$$

$$U'' = L - N'' = L - (L.N'')N''$$

Solving further, we can substitute the values for calculating reflection vector *R* as:

$$R = (L.N)N - (L - (L.N)N$$

$$R = (L.N)N - L + (L.N)N$$

$$R = 2(N.L)N - L$$

Figure 16: Phong Shading with shininess power of 128.

2.2.5 Blinn-Phong Shading:

Phong model is efficient way for approximation of lighting. However, it's specular calculation runs into a problem. When the angle between reflection and view doesn't go over 90 degrees, the specular portion gets cut off. Also, one must continually recalculate the dot product *R.V* between view direction *V* and reflected vector *R*.

This problem was solved by James F. Blinn [11] which tells that instead of creating a reflection vector *R* using reflection formula, it can be calculated using the sum of viewing vector *V* and light vector *L* and normalizing it giving us the two-way vector. This is also known as halfway vector where the unit vector is exactly half between view direction and light direction. As a result, the dot product between surface normal and halfway vector will never overshoot 90 degrees. This halfway vector *H* is calculated as:

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

2.3 Lighting in modern video games

2.3.1 Physically Based Rendering:

While Phong or Blinn-Phong shading does improve the visual quality of the image frame, it is still not close to being realistic. Physically based rendering (PBR) is combination of different rendering techniques that performs lighting calculations that closely matches the real world. It was introduced by Yoshiharu Gotanda at SIGGRAPH 2010. PBR also makes the work of the artists a lot easier as they no longer use any kind of hacks like increase the colour range more than 1.0 and other stuff.

PBR works on the microfacet surface model which means that the surface normals of any object are not uniform or smooth and it contains rough surfaces when magnified.

It also promotes energy conservation, which means the amount of reflected light is always the same as the incoming light except for surfaces with emission. [9] explains the PBR pipeline in more details.

2.3.1.1. Microfacet model:

Microfacets are little perfect mirrors on a microscopic scale. The alignment of these microfacets depends on the roughness the surface. More the roughness, the light will reflect in random direction depending on the microscopic surface normal. Since, microscopic surface normal is not possible on per-pixel basis, a simple roughness parameter in the range of 0.0-1.0 is introduced to average the roughness of the object. The ratio of the microfacets can be calculated from the halfway vector formula H explained in Blinn-Phong shading above.

Where L is the light direction and V is viewing direction. If the microfacet is aligned close to then it will create strong specular reflection or a bright highlighted spot.

2.3.1.2. Energy Conservation Law:

When the light falls on the surface of object, it either gets reflected which means the light rays clearly bounces off and never enters the surface or it gets refracted meaning some portion the light is absorbed by the surface. The light that gets reflected is called Specular Reflection and the one which gets absorbed is called Diffuse Reflection.

In real physics, the diffuse light reflection is not always absorbed, but rather the refracted light is bounced inside, where some lights may get reflected out and some don't. This however depends on the type of the surface the light is interacting with. If the surface is completely metallic, then the diffuse light is immediately absorbed and only specular light is reflected. In computer graphics PBR pipeline, it introduces a new parameter along with roughness which is called Metallic, and it also ranges from 0.0-1.0. Pure Metallic surfaces will never show diffuse colours.

2.3.1.3. Rendering Equation:

The rendering equation was explained in brief in introduction section. It is an integral equation used in computer graphics for determine how exactly light interacts with surfaces and makes sure how the final object will look like after light falls on it. The equation looks like:

$$Lo(p, \omega o) = \int_{\omega i} F(N, V, L) Li(p, \omega i) cos\theta d\omega i$$

Here, *L* denotes the term called Radiance which explained before. To summarize again, Radiance is used to measure the strength of the incoming light from the single direction. A term called Radiant Flux is used to measure the energy sent from a light source in Watts.

In real life, light is sum of energy over a different wavelength graph. Each wavelength gives a particular colour in nanometers with ideal range between 390nm and 700nm. Taking these wavelengths for computer graphics in not possible.

ω here denotes the solid angle. The solid angle tells the size of shape which projected on a sphere with radius of 1. See Figure 17.

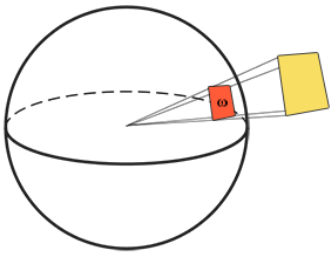

Figure 17: Solid Angle.

Radiant intensity is the intensity of the light source over a projected area of the unit sphere. It is calculated using formula:

$$I = \frac{d\Phi}{d\Omega}$$

Here, *I* is the radiant flux divided by solid angle. With all this, the final radiance *L* can be calculated using the formula:

$$L = \frac{d^2\Phi}{dAd\omega \cos\theta}$$

Here, *A* can be represented as point *P* of the fragment in world space and solid angle ω can be represented the light direction vector and cosine angle θ is the dot product of the light direction and surface normal *N*. So, *L* here denotes the radiance at point *P* in direction solid angle.

From the rendering equation, *Lo* is the irradiance which is the sum of all the radiance of all the light sources.

2.3.1.4. Bidirectional Reflective Distribution Function (BRDF):

BRDF is also explained in introduction section. To summarize again, PBR uses a special Bidirectional reflective distribution function or BRDF. In computer graphics, this reflective property is called Roughness/Glossiness/Smoothness in some 3D engines. The BRDF function requires three inputs and is denoted as *F(N, V, L)*. Here, *N* is the surface normal, *V* is the direction from camera or eye to surface and *L* is the direction of light to surface.

The most famous BRDF is the Cook-Torrance BRDF which is given by the equation:

$$f = Kd\ flambert + Ks\ fcook - torrance$$

Where, *Kd* is the light ratio that will get refracted and *Ks* the light ratio that will be reflected.

2.3.1.5. Diffuse BRDFs:

Reflections comes in two types: Diffuse and Specular. Diffuse is a reflection where photons enter the surface and moves out after many bounces. Whereas the Specular get reflected or refracted. See Figure 18.

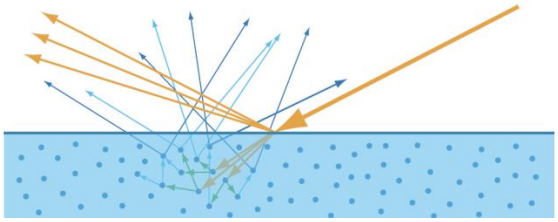

Figure 18: Yellow- Specular reflection, Blue- Diffuse reflection.

To start with diffuse BRDFs, the most common the Lambertian diffuse BRDF which is given by the formula:

$$flambert = \frac{C}{\pi}$$

Here, *C* is the simple albedo colour which can be obtained from any albedo texture and is divided by π for normalizing the reflected diffuse light or in other words satisfy the energy conservation law. There are other types of diffuse BRDFs like the

Oren-Nayar BRDF, Disney Burley BRDF which will be explained after PBR and IBL.

2.3.1.6. Specular Cook-Torrance:

The specular BRDF is the Cook-Torrance Specular BRDF which is again given by the formula:

$$fCook - Torrance = \int Ks \frac{DFG}{4(wo\ .n)(wi\ .n)}$$

It is made from three sub-functions; D is the Normal distribution function (NDF) which approximates the alignment of microfacets along the halfway vector h based on the roughness parameter. Here, Trowbridge-Reitz GGX is used to calculate NDF using the formula:

$$NDF(n,h,r) = \frac{r^2}{\pi((n.h)^2(r^2-1)+1)^2}$$

Here, n is the surface normal, r is the roughness value and h is the halfway vector between the surface normal and light direction. If the roughness is zero, it will create a bright spot and vice-versa as it seen in figure 19.

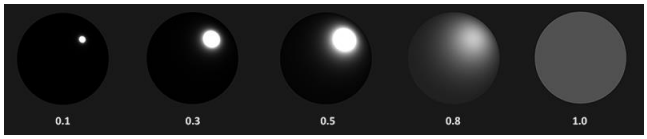

Figure 19: NDF Reflections based on roughness strength.

2.3.1.6.1. Geometric Shadowing Function:

The second sub-function is Geometry function whose purpose to calculate the shadow factor of the microfacets which again depends on the strength of the roughness or glossiness. Here, the Schlick-GGX formula is used with normal, view and roughness as inputs.

$$Gschlick - GGX(n,v,k) = \frac{n.v}{(n.v)(1-k)+k}$$

Where, k is remapping of roughness r based on direct or image-based lighting. For doing approximation of geometry, the Smith method is used for with Schlick-GGX which takes light direction vector l and view direction vector v and remapped roughness k being common for both. Figure 20 shows the Schlick-GGX method with different roughness strength.

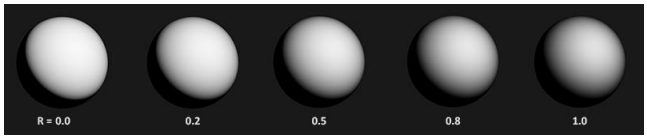

Figure 20: Geometry Shadowing based on roughness strength.

There are other G functions like Neumann, Keleman for approximations of shadowing.

2.3.1.6.2. Fresnel Function:

The third and last sub-function is the Fresnel which is most important function because in real life everything has shine when looked from a certain angle. Even a real-life brick object with highest roughness strength will show some amount of specular effect when looked at almost 90-degree angle. The Fresnel-Schlick formula is the most common approximation used in films and video games which has its formula:

$$Fschlick(h,v,F0) = F0 + (1-F0)\big(1-(h.v)\big)^5$$

One good thing about Fresnel-Schlick approximations is that it takes account of both metallic and non-metallic or dielectric surfaces. Just like roughness strength of range 0.0-1.0, game engines also make use metallic strength which defines if the surface is pure metal or dielectric with 1.0 being pure metal and 0.0 being dielectrics.

F0 is the base reflectivity of the surface which is achieved using the Index of Refraction (IOR) depending on the type of surface of object like metal, water, plastics, etc. Fresnel is seen more when the angle between the view vector and halfway vector reaches almost 90-degrees. See Figure 21.

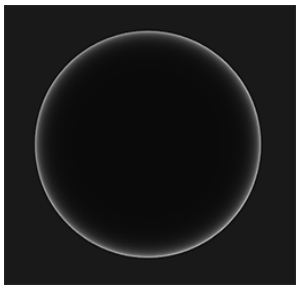

Figure 21: Fresnel is seen at angle reaching at 90-degrees.

Unreal Engine 4 [10] uses a modified Schlick approximation which uses the Spherical Gaussian approximation to replace the power of 5 in the above formula which gives minor performance boost.

$$Fschlick(h,v,F0) = F0 + (1-F0)2^{(-5.55473(v.h)-6.98316(v.h))}$$

After adding all the calculations of Lambert and Cook-Torrance BRDFs, we get nice realistic approximation of real-world lighting using light sources as shown in Figure 22.

Figure 22: PBR with different light sources.

While Cook-Torrance is one specular model, there are other specular models that can be integrated in the PBR pipeline.

2.3.1.7. Specular Gaussian Distribution:

One of the other specular models that can be used for normal distribution is the Gaussian Distribution function which is also called as classic "bell-shaped" curve distribution for calculating the probability density of the distribution at point X. It is given with the equation:

$$f(x) = \frac{1}{\sqrt{2\pi a}} e^{-\frac{(x-u)^2}{2a^2}}$$

The e in the equation is a constant to approximately 2.718 and u is the average mean. The $\boldsymbol{a^2}$ is the variance of the gaussian distribution. Larger variance value will result in flatter and wider distribution of microfacet normal.

In computer graphics sense, the surface normal is the average and x is the angle between surface normal between halfway vector. The above equation can be rewritten as:

$$f(x) = e^{-\left(\frac{a}{m}\right)^2}$$

Here, *m* is the roughness strength in the range of 0-1, with larger values represent rougher. Also, *a* is the actual angle between surface normal *N* and halfway vector *H*. By actual angle means not the dot product or cosine angle but angle in radians that can be obtained using inverse cos of dot product to get the angle. Figure 23 shows the Gaussian Specular in action.

Figure 23: Gaussian Specular [*a* = 0.3] (Pale specular highlight on front car window).

Roughness strength value of 0 tends to leave a small solid and tight looking highlight instead of leaving a smooth tail as it moves/distributes further away from the actual spot.

2.3.1.8. Specular Beckmann Distribution:

Gaussian specular is good microfacet distribution but it was not physically based and requires a very high exponent value. One physically based approximation is the Beckmann-Spizzichino distribution [12]. As mentioned, that this function is physically based, so it will require more computations compared to Gaussian and it is derived by the isotropic equation:

$$f(x) = \frac{\exp\left(-\tan^2(a)/m^2\right)}{\pi m^2 \cos^4(a)}$$

$$a = \arccos\left(N.H\right)$$

Where, *a* is angle in radians of normal *N*, halfway *H* and *m* is which the root mean square defined as square root of mean squared. As a result, the roughness is remapped as:

$$m = \sqrt{2a}$$

Again, the denominator contains π to avoid the surface emitting out more light energy. The equation provided requires more computation and as a result, the model can be modified as:

$$\tan^2\frac{a}{m^2} = \frac{1-\cos^2(a)}{\cos^2(a)m^2}$$

Just like Gaussian specular, this type of distribution returns a wider and tight tail-less looking specular highlights and it makes a perfect candidate for making perfect clean mirror like reflections with roughness strength of 0. Figure 24 shows the result with strength of 0.3:

Figure 24: Beckmann Specular.

2.3.1.9. Specular GGX Distribution:

The full form of GGX is unknown. It is an extension of the Beckmann Specular distribution with aim to provide stronger and wider tails. It was proposed by Walter, Marscher, Li and Torrance [13]. It is a generalization of Trowbridge-Reitz distribution which is also common in computer graphics and is also defined by the roughness parameter alpha. It has properties such as smooth transition from Isotropic to Anisotropic cases. The authors have done some test cases like ground, glass materials, and compared with other distribution like Phong and Beckmann distributions and concluded that the results were a closer match to real world.

The GGX distribution formula is given as:

$$D(m) = \frac{a^2(m.n)}{\pi \cos^4\theta m\left(a^2 + \tan^2\theta\right)^2}$$

Where *m* is the microfacet surface normal and *n* is the surface normal θ is the angle between surface normal *N* and halfway *H* and lastly *a* represents the roughness parameter. As usual, the π is used to obeying the energy conservation law. The *tan* θ could require more computations as it squared, hence the distribution function can be simplified for graphics as:

$$D(m) = \frac{a^2}{\pi\,(\cos^2\theta\,(a^2-1)+1)^2}$$

The result is almost like Beckmann or Phong distributions if roughness is set to lower value but gives more tail similar in real world. Figure 25 shows the GGX output with roughness 0.3:

Figure 25: GGX Specular.

2.3.2 Image-based Lighting (IBL):

The project also makes use of cubemaps for rendering a skybox, but simply rendering a skybox is not enough. In real life surrounding environments provide light data which is good for ambient lighting. Hence, this project has implemented and demonstrated Image-based Lighting (IBL) which is used for ambient colour lighting. The project used five different HDR format skyboxes which are loaded and converted to 6-sided cubemap and other algorithms are run to get diffuse and specular reflection from the image. IBL is a technique for treating the environment as light sources and is important part in PBR pipeline as makes the object look physically accurate. All these skyboxes are pre-computed at the start of first frame of application.

2.3.2.1 Diffuse Irradiance IBL:

From the rendering equation of PBR, the diffuse Lambert and π are constant and so to solve inner integral variables, large number of incident light rays coming are texels are sampled in the hemisphere and stored in a another cubemap which is called an Irradiance map which looks like blurred version of the original cubemap.

The HDR map cannot be used directly used as a cubemap as it is stored in equi-rectangular or sphere-rectangular format. The first step is to convert the points on sphere to UV space and using those UV coordinates can be projected on a cubemap. The job of UV space is to bring those sphere radians PI into texture coordinates range of [0, 1]. For mapping Sphere to UV space, the sphere is projected onto a cylinder border and is cut from one side to unwrap the cylinder. Having points on sphere, theta and phi are found using trigonometry functions like atan and asin. So, for U and V the formula is given as:

$$U = 0.5 + \frac{\arctan(dx, dy)}{2\pi}$$

$$V = 0.5 + \frac{\arcsin(dy)}{\pi}$$

With this, an original cubemap is created from the spherical map and now the important step to make the irradiance map. Since, a cubemap is combination of 6 different sides, so 6 different snapshots need to be created for doing convolution. For this, I setup six different cameras or view-matrix each facing one side of the cube and using a Framebuffer to take a snapshot and store it in the frame buffer's colour component texture which is then sent to the fragment shader for convolution process.

The convolution process is done by taking average of an area of hemisphere. The area gets smaller as the process moves towards the center-top of hemisphere. After the convolution step is done all the fragment data in stored in another cubemap with dimensions being 1/4th dimension of original cubemap. To apply this, convolution in ambient lighting, a modified Fresnel-Schlick equation is used which makes of roughness as the original one requires a halfway vector which is not possible to get from hemisphere. Hence, a direct roughness value is sent to it. The Fresnel-Schlick roughness equation looks like this:

$$Fschlick(n, v, F0, r) = F0 + \max(1 - r, F0) - F0\big(1 - (n.v)\big)^5$$

Where *n* is surface normal, *v* is viewing direction, *r* is roughness of the surface. The reason is to simulate some amount of Fresnel effect at grazing angles and because it is a necessary part of PBR pipeline.

2.3.2.2 Specular IBL:

From the Cook-Torrance microfacet equation explained above:

$$fr(p, wo, wi) = \int Ks \frac{DFG}{4(wo\,.n)(wi\,.n)}$$

It is seen that the term *Ks* is not constant in the integral and is dependent both on incoming light direction as well as view direction. Solving this integral for all possible light direction and view directions per frame is expensive. Hence, the pre-computing step also takes place here just like the irradiance map. A solution is proposed by Epic Games where the pre-computing of specular map is done in two parts. The two parts are pre-computed and later added together which is called as Split-Sum approximation. The two part equation integral looks like this:

$$Lo(p, wo) = \int Li(p, wi)dwi * \int fr(p, wo, wi)n.wi\ dwi$$

2.3.2.2.1 Split-Sum First Part:

The first part is like irradiance step with additional roughness strength considered. The convolution happens by storing different mip levels of the cubemap and moving between them depending on the roughness level. All this convolution data is stored in another cubemap called a Pre-filtered cubemap with mipmapping option on so than OpenGL will make mipmaps. Since, view and reflection direction is not taken into account here it assumed that view direction is same as output direction *wo*. However, there is a catch that specular reflections cannot be seen at grazing angles.

The base mip level dimension of the cubemap starts at 128x128 which is used when roughness value of 0 and dimension is cut in half when moving up the mip level for increasing roughness. For more accurate reflections, the base dimensions can be increased at the cost performance.

For generating the specular lobe, when light radiance falls on surface, it reflected or scattered around the reflection depending on roughness level. To pre-compute this scattering of radiance data, a process called importance sampling is used where large random set of data is generated using Monte Carlo integration and is discretely sampled. The sequence of data is created using Hammersley's sequence so that the sequence is evenly distributed.

The sampling process is done by taking a large sample count and running a loop. Inside the loop, random sample value is created which is used to create sample vector in tangent space and is later converted in world space and lastly sample the scene's radiance.

2.3.2.2.2 Split-Sum Second Part:

The second part is doing the actual BRDF in specular integral. This BRDF response is also pre-calculated using the inputs roughness and angle between surface normal and light direction and the output of BRDF is stored in a lookup texture of two channels – Red and Green, which is as called BRDF LUT where X-axis stores the angle and Y-axis stores the roughness. The green colour gives the Fresnel response.

The convolution process is very similar to the first part by using the Hammersley's sequence to generate sample vector. The only difference here, since it's a BRDF which contains Geometric function and Fresnel. So the sample vector is processed using the same Schlick-GGX function for G and Fresnel-Schlick for F. The two output is scale and bias factor F0 from each sample is averaged at the end. Figure 26 shows example of BRDF LUT generated.

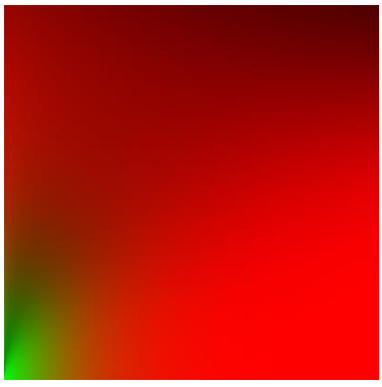

Figure 26: BRDF Lookup Texture.

After calculating the BRDF and filtering it, the final step is to add both the parts in the final split-sum approximation and the results can be viewed in Figure 27. In the figure, there are no additional light sources like point, directional and lighting data is coming from the cubemap. Notice the ground is receiving light from the palace in background.

Figure 27: Image-based Lighting. Ground is receiving light from building in cubemap.

2.3.3 Disney BRDF:

The Disney PBR deserves its own section as it another huge topic. It was introduced in by Brent Burley [19] at the SIGGRAPH 2012 conference and was introduced as a BRDF and used in animated movies like Wreck-It Ralph. The BRDF was further extended into a BSDF with the support of sub-surface scattering in the diffuse part of when how light enters the surface.

2.3.3.1 MERL 100:

While the BRDF is physically based, it was made to fit all the materials in the MERL 100 [14] database which was capture by Matusik et al. which contains the isotropic BRDF functions of all different materials like paints, metals, rubber, plastic, etc.

To try out different BRDFs, Disney made a free open-source tool called BRDF explorer to examine and analyze different BRDFs using OpenGL and UI support for change different parameters. The BRDF explorer used a technique called Image Slice which are group of images viewed at different angles for analyzing and check if it matches with material from MERL 100.

2.3.3.2 Principled BRDF Parameters:

The name principled, means the model doesn't strictly need to be physical based and can be changed to something else depending on the art directions. The model came basic as well as new parameters listed below:

- Base Colour: The albedo or surface colour
- Subsurface: Controls the subsurface scattering
- Metallic: Controls if material is dielectric or metallic
- Specular: The specular control
- Specular Tint: Tints specular towards base colour
- Roughness: Controls the smoothness of material
- Anisotropic: Controls the aspect ratio of specular

- Sheen: Primarily used for cloth materials
- Sheen Tint: Sheen tint towards base colour
- Clear Coat: Another addition of specular
- Clear Coat Roughness: Controls the clear coat glossiness

2.3.3.3 Burley Diffuse BRDF:

The diffuse BRDF they used to be a modified lambert diffuse which uses the Fresnel factor with roughness in account. The reason for this they mentioned is due to lambert causes dark colour around the edges. This diffuse BRDF is also sometimes called as Burley Diffuse and it uses same Fresnel-Schlick function with retroreflection function in order to avoid dark edges. The whole diffuse equation is given as:

$$fd = \frac{C}{\pi}(1 + (Fd90 - 1)(1 - cos\theta l)^5)(1 + (Fd90 - 1)(1 - cos\theta v)^5)$$

$$Fd90 = 0.5 + 2\,roughness \cos^2 \theta d$$

Here, Fd90 is the retroreflection that produces diffuse reflectance of 0.5 at grazing angles and goes up 2.5 for extreme rough surfaces. It was also fairly matching with the MERL data.

The subsurface scattering in diffuse reflectance they mentioned makes use of Hanrahan-Krueger subsurface BRDF [15] and the subsurface parameter blends between Burley and HK BRDF. When using this parameter some minor differences can be noted.

2.3.3.4 Specular BRDF:

For normal distribution function D, Burley used the Generalized Trowbridge-Reitz or GTR for both isotropic and anisotropic. GTR results are almost matching with results of Beckmann when the exponent value is 10. Instead of one, his specular lobe contains two lobes both using GTR model. The first lobe is for base material that can be anisotropic and metallic whereas, the second lobe is only isotropic and non-metallic which is handled by Clear Coat parameter.

The anisotropic is used to show specular effects on surfaces like brushed metals where the normal are not evenly distributed. The Anisotropic model their model uses is GTR2. The anisotropic model is dependent two weights *ax* and *ay*:

$$ax = \frac{roughness^2}{aspect}$$

$$ay = roughness^2 * aspect$$

The aspect of the specular is calculated using the square root of anisotropic parameter:

$$aspect = \sqrt{1 - 0.9 * anisotropic}$$

The final GTR2 anisotropic distribution function is given as:

$$DGTR2 = \frac{1}{\pi axay} \frac{1}{\left(\frac{(h.x)^2}{ax^2} + \frac{(h.y)^2}{ay^2} + (h.n)^2\right)^2}$$

Here, *x* and *y* are the tangents and bitangents of the surface normal *n* and *h* is the halfway vector between light direction and view direction.

For Fresnel function F, it still uses same Fresnel-Schlick already explained in PBR section. The last Geometric Shadowing function G used is the same Smith-GGX function also explained before, but minor change of roughness being brought to range of [0.5, 1] instead of [0, 1]. The reason for this change was to match with MERL data and feedback given by Disney artists. The clearcoat parameter uses the different Smith G function with constant roughness of 0.25 which was found to be aesthetic.

Figure 28: Disney BRDF with Anisotropic Specular. Notice specular effect as stretched line.

Overall, the Disney BSDF provides visually appealing effects with wide range of parameters and two specular lobes on top of it. Performance wise, it is still quite heavy due to more parameters which make not suitable for applications where performance is top priority.

2.3.4 Oren-Nayar Diffuse BRDF:

The last and final BRDF I wanted to mention is Oren-Nayar [16] diffuse only BRDF which also a physically based and works on microfacet level using roughness as parameter. The model works on assumption of isotropic surface and is a solution for Lambert diffuse problems where dark edges are seen at grazing angles and doesn't take roughness of surface into account. It uses the roughness model proposed by Torrance and Sparrow which tells that surfaces at the micro surface level have a V shaped cavities with upper edges lying on same plane. Each cavity consists of two planar facets and the roughness of the surface is specified using a probability distribution function for distribution of facet normal. Figure 29 shows the V shaped cavities:

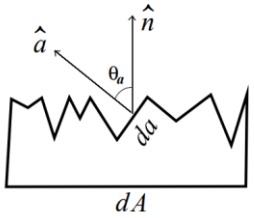

Figure 29: Micro surface with V-cavities.

The probability distribution function determines the final roughness of the surface. The orientation and slope of each microfacet model is shown as θa and θb. Torrance-Sparrow uses the distribution $N(\theta a, \theta b)^4$ to get the surface area of the single microfacet. Probability distribution function is referred to as Slope-area distribution as they have mentioned is easy to use. They experimented with different types of surfaces with different types of slope distribution:

- Unidirectional Single Slope distribution: Results in non-isotropic where each facet has same slope and cavities aligned in same direction.
- Isotropic Single Slope distribution: Where all facets have same slope and are evenly distributed.
- Gaussian distribution: Slopes are assumed with mean zero and standard deviation tells the roughness of the surface.

2.3.4.1 Roughness Reflectance:

Oren and Nayar explained an example working of what happens to a V-cavity when a light source emits to it in relation to view/eye position. They wanted to prove the point that rough surfaces are non-Lambertian in real life. Both facets of V-cavity are receiving light energy from right side at a distance. If the facets are Lambertian, then the left sided facet will receive more brightness than the right facet. Also, if the eye position is seeing the cavity from the left side, that large portion of the facet will be seen darker and remaining smaller portion will be seen bright.

If the eye position is set on the right side as the light source position, then opposite happens where large area of the facet will be bright and small portion appears dark. To conclude, the facet's radiance increases when eye and light direction are nearly same. See Figure 30.

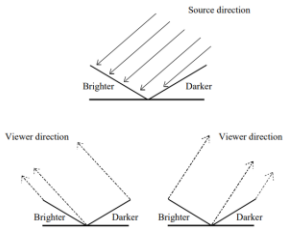

Figure 30: Radiance of V-Cavity increases are viewer reaches light source.

2.3.4.2 Facet Radiance and Irradiance:

The Irradiance in their model is calculated by doing the average all radiances that facets receive with the help of Slope-area distribution function. For calculating the radiance that a single facet will receive is done by having surface area *dA* that covers large quantities of V-cavities with similar slopes. Each V-cavity has two facets facing away from each other in opposite direction. If flux emitted by facet with surface area *dA* and normal of microfacet is *a* (Figure 28), then the projected surface area is:

$$Proj\ dA = dA\ Cos\theta a$$

This projected surface area is needed to find out the contribution of facet that it gets the radiance. Lastly, the final radiance measure is calculated as:

$$Lrp(\theta a, \theta b) = \frac{d^2\theta r(\theta a, \theta b)}{(dA\ Cos\theta a)Cos\theta r\ d\omega r}$$

For getting the Irradiance, the slope-area distribution function used is $P(\theta a, \theta b)$ and mean of all radiance $Lrp(\theta a, \theta b)$. Hence, the final irradiance equation is:

$$Lr(\theta r, \theta i) = \int_{\theta a=0}^{\pi/2}\int_{\theta b=0}^{\pi} P(\theta a, \theta b)Lrp(\theta a, \theta b)sin\theta a\ d\theta b\ d\theta a$$

2.3.4.3 Facet Interreflection:

The diffuse model also has support for interreflections where light rays bounce from facets. If there are no interreflections then radiance of facet it receives is null and if the angle of facet normal and light direction is large enough and the viewing angle is from other side, then facets won't be seen. Therefore, light bouncing from neighbouring facet will receive some amount of radiance. In order to avoid infinite light ray reflections, only two light reflections are done.

2.3.3.4 Geometry Shadowing and Masking:

If the facets are viewed from the surface normal angle, then all facets receive radiance and are illuminated. However, for larger angles of incidence facets are shadowed by other facets that are in front of them. Shadowing occurs when a facet is partially illuminated when its neighbour is casting shadows on it and Masking happens when adjacent facet obscures the facet and is barely visible to the viewer. These two terms directly affect the projected radiance of the facet. For calculating this geometric shadowing, the authors made a function called Geometric Attenuation Factor (GAF) that outputs the range of [0, Infinity] which helps to reduce the projected radiance of the facet and explained that it equals the ratio of visible and illuminated facet surface area to total surface area of facet. The GAF function is also correct for any facet surface normal. Its final formula is given as:

$$GAF = \min\left(1, \max\left(0, \frac{2(l.n)(a.n)}{(l.a)}, \frac{2(v.n)(a.n)}{(v.a)}\right)\right)$$

Where, *l* is the incident light direction, *n* is surface normal, *a* is the microfacet surface normal and *v* is the view direction towards the surface normal.

The authors have tested the experiments with different objects by taking the real-world images of it and comparing it with their method. They also have created model with different distribution functions. Oren-Nayar model today is widely used by the computer graphics community especially in applications for rendering rough surfaces.

3 DESIGN AND IMPLEMENTATION

All the related works explained and discussed in the above section are demonstrated in the project I have developed along with additional features I have implemented is explained in this section.

3.1 Tools and APIs Used:

3.1.1 OpenGL:

OpenGL is an open-source graphics API developed by Khronos Group and is still widely used in the industry. The decision was made to use this API because it is quite easy to setup and run. Although its state-machines like behaviour is old and I have quite some experiences using it. Also, OpenGL is still very big with lots of API and features inside that can be utilized for better efficiency and performance of the application.

3.1.2 Visual Studio:

For programming IDE, the most popular and latest Visual Studio 2022 is used and the programming language is C++ so that I could use the C++ 11 features.

3.1.3 NCLGL Framework:

The NCLGL framework provided by the university professors is being used as a base and it has been extended with other abstract classes required for this project and it uses Win32 for creating windows. It also comes with ready-made event system for handling inputs.

3.1.4 Image Loading:

There are quite many free image loading libraries available out there. Previously, NCLGL framework provided was using the Simple Open Image Library (SOIL), but there were some problems with it like the image inversion in the Y-axis was taking long time and no options were there to provide desired texture format. After finding out that SOIL uses old version of STB image library. I decided to remove SOIL and add latest version STB image loading library developed by Sean Barrett which is free and open source. It supports most of the known image formats like PNG, JPEG, TGA, HDR, etc.

3.1.5 Dear ImGui:

Changing parameters, compiling, checking results, exit and repeating the same steps can be annoying. To see the results immediately at runtime, a popular GUI tool is integrated in this project. Dear ImGui is a free graphical user interface library which is useful for any 3D graphics applications. It helps programmers to visualize and create debug tools. Its widget system allows for easy creation of UI like buttons, dropdowns, etc. It also comes with its own input system. This project uses the special docking version of ImGui that enables seamless experience of docking windows and properly organize them.

3.2 Shaders:

3.2.1 Shader Class:

The main shader class is modified which now includes additional methods for sending data to the main GLSL shader classes. This helps to avoid calling all big OpenGL functions like glUniforms, glUniformMatrix, etc which prone to calling wrong functions or wrong data can be sent. It has methods for respective uniform datatypes like integer, float, vectors, texture and texture cubemaps.

The class also contains a hash map which stores key as uniform name and uniform location ID made by OpenGL as value. So whenever a method is called like SetInt() for example which has parameters uniform name and the actual integer data that will be sent to the GLSL shader. It first checks if the uniform is already included in the hash map. If it does, simply return the uniform ID. If is not included, then get the uniform location from GPU and save it in the hash map. This minor improvement gives performance benefits as we don't have to call get uniform location function again.

3.2.2 GLSL Shaders:

The project contains many shaders, but the main BRDF shaders I have made for lighting are:

- PBR Cook-Torrance BRDF
- PBR Disney BRDF
- Blinn-Phong BRDF
- Oren-Nayar BRDF

The Oren-Nayar BRDF also contains three additional specular BRDFs namely Gaussian, Beckmann and GGX.

The current BRDF shader can be easily changed at runtime using ImGui tool. All the internal workings of these BRDFs are already explained the Related Work section.

3.3 Material System:

The material system is not directly related to real-time lighting. I wanted to create a very basic material system for my own personal knowledge to learn how the material system works inside the graphics engine. This was inspired by the current generation of game engines like Unity, Unreal engine. From what I learned is a Material contains a set of data like colour, some scalar parameters and can be applied to a single mesh or to sub-meshes as well. In this project, I have not implemented materials for sub-meshes and so it only applies to the whole mesh.

The datatype of material is a structure. The master base structure contains single parameter that stores colour and four additional material structs are derived from master material structure. As explained in shader BRDFs section above, these four additional structs store data for respective BRDF shaders.

3.4 Post Processing Effects:

Post processing really interests me a lot because it gives the power to final image to something completely different. For this reason, I have done four types of post processing effects i.e., Bloom, Screen Space Ambient Occlusion (SSAO), Vignette and Invert Colour. The post process renderer class is responsible for taking care of all different effects. Each effect has a type which tells if the effect should do post processes pass before lighting (Pre), after lighting (Mid) and final image (Last).

3.4.1 Screen Space Ambient Occlusion (SSAO):

SSAO was first introduced by Crytek for CryEngine 2 [17] for showing soft dark corners around edges of the object using only depth buffer. I have used this effect as pre-pass meaning it is done before any lighting calculations is done. A framebuffer is used with only depth texture attached that will render the whole scene and store in a single channel depth texture. After this, the texture is sent to the SSAO shader what will do the computing. SSAO works by converting the fragment from screen space to world space of the depth texture and is compared with other fragments inside the hemisphere. If the fragment is occluded, then it is given a dark colour. A simple blur process is run to make it smoother. The final SSAO texture is then stored and sent to any lighting shaders and merged during the ambient lighting calculations along with ambient occlusion texture of the object if it has any. Figure 31 shows the raw SSAO effect.

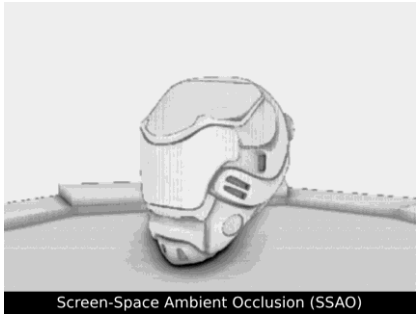

Figure 31: Screen Space Ambient Occlusion.

3.4.2 Bloom:

Bloom always fascinates and makes the scene look beautiful if done right. Physics based bloom effect was introduced by Sledgehammer games and shown in Call of Duty: Advanced Warfare game [18]. It is done in series of steps. All these steps are done with the help of framebuffer with only colour attachment. The first step is to filter out all the less bright colours and make them completely black if their brightness is less than threshold value. Again, a special framebuffer is created called FrameBufferBloom which stores an array of textures called mipchain which will store textures data of different mip level. Maximum mip level is 8.

The main stage consists of two parts: Downsample and Upsample part. The downsample part is done by scaling down the image by half at each stage which is run in fixed iteration. Then a special blur kernel developed by developers at Sledgehammer is used on each downsample. The upsample is reverse of downsample where texture is upscaled twice the original and added with the blurred downsample until texture dimension matches the original. The final filtered and sampled texture is then merged with original colour texture obtained from framebuffer running another simple shader. Result is shown in Figure 32.

Figure 32: Bloom.

3.4.3 Vignette:

Vignette effect is common in photography where borders of the image are darkened. Fragments further from center of the image are darkened creating a tunnel vision effect. A simple vignette effect is created and is merged with final colour image. Figure 33 shows the effect.

Figure 33: Vignette.

3.4.4 Invert Colour:

This is the simplest post processing effect that inverts colour of final image. See Figure 34.

Figure 34: Invert Colour.

3.5 Billboards:

This is not related to lighting in the project. Billboard is a simple flat quad mesh which will always face the camera or any target it is given. The direction always changes in runtime when the camera or the billboard object moves. Referring to [6], I will explain the type of billboards and its implementations. Billboards comes in three types:

3.5.1 Point Sprites:

Point Sprites billboards are common billboards where the center pivot point is in the middle of the quad mesh and is rotated around that pivot point. The current project utilizes the point sprites billboards.

To implement the point sprite billboards, it requires special billboard matrix, which is like a normal model matrix, but for the billboards.

The first step is to get the look direction vector from billboard to camera which is obtained by simply subtracting the camera position and billboard position in world space and normalizing it. This look vector is required for any type of billboard.

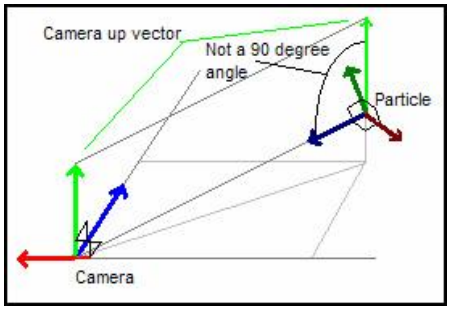

Figure 35: Point Sprites.

After getting the look vector, we need to calculate the right vector of the billboard. For this, a temporary camera up vector is required and simply doing the cross product of camera up and look vector will give the right vector of the billboard. Lastly, to get up vector of billboard, we again do the cross product of the look vector and right vector we got in previous step.

This billboard type can be seen in game engine as gizmos like Unity, Unreal, etc.

3.5.2 Axis Aligned:

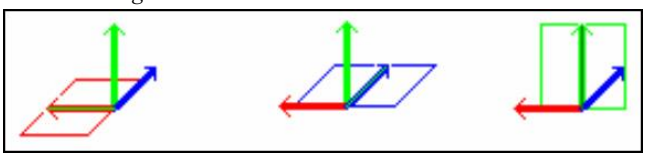

Figure 36: Axis Aligned Billboards locked Y axis.

Axis aligned billboard is a lot simple as it doesn't the temporary camera up vector. AA billboards always rotate around the locked global Y axis or the up axis. In OpenGL it is a simple up vector of (0, 1, 0). The look vector works as forward vector and right vector calculated by doing the cross product of the look and up vector.

This kind of billboards can be seen in action in old games like Wolfenstein 3D and Doom.

3.5.3 Arbitrary Axis:

Arbitrary axis billboard works slightly different. The look vector here is not the final look vector, but a temporary one which is used to get the right and up vectors of the billboards. This temporary look vector is not orthogonal to the up vector. The up vector is given by the axis we want to rotate around which can be either X, Y or Z.

To get the right vector, a cross product of the up and temporary look vector is done. Finally, the original look vector can be calculated using the cross product of up and right vector. See figure 37.

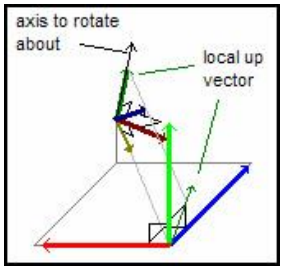

Figure 37: Arbitrary Axis.

3.5.4 Billboard Matrix:

Once the entire forward, up and right vectors are done, the billboard matrix can be calculated and is sent to the vertex shader. To calculate the matrix, the three – forward, up and right vectors are put in a 4x4 identity matrix. The billboard matrix looks like this:

$$\begin{bmatrix} R1 & U1 & F1 & Px \\ R2 & U2 & F2 & Py \\ R3 & U3 & F3 & Pz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where R is right vector, U is up vector, F is forward or look vector and P is the position of the billboard in world space. This

matrix is later multiplied with the view projection matrix in the vertex shader.

3.6 Uniform Buffer Objects:

Having discussed the approach, I have implemented the forward lighting in the fragment shaders, but this time with small changes. Previously, I used to send all the lighting relevant data via their uniforms location every frame to the GPU, but now after learning the use of Uniform Buffer Objects (UBOs), I don't have to send the lighting data every frame.

With additional integration of Dear ImGui tool which is an awesome tool for debugging or changing a value at runtime. Whenever, any property of a light is changed using ImGui, only a small block of the whole UBO light object is changed by providing the offset and size of the data that was changed. This small optimization helps.

Multiple light sources like point, directional and spotlight are also implemented with maximum amount of 100.

The same UBO object is also used for storing the projection and view matrix of the camera object instead of sending then every frame.

3.7 Textures:

Texture class was created to automatically get the format of texture is loaded like RGBA, RED, etc. and get its width, height dimensions of the image whenever required along with options like what internal format to give before loading the image. With this I have also created more texture classes which inherits from normal texture class like HDR texture class for HDR and gamma correction and a Cube Map texture class for converting the equirectangular HDR image to a 6-sided cube map texture.

3.7.1 Using Threads:

One challenge was to reduce loading times of texture file size due to being large as all the textures were loaded using main thread and hence, they are loaded one by one synchronously. Normally, for using multi-threading in OpenGL requires to make multiple contexts with sharing data between contexts. However, due to time constraints this couldn't be implemented and small workaround I did was using threads provided by C++ which will load only raw data from the texture file and after all the threads are done doing their job, the raw data are used to OpenGL texture objects ready to be sent to GPU.

3.8 Tone Mapping:

For Tone Mapping to work, the colour texture format attached to the frame buffer need to be of 16-bit or 32-bit float format for it work. Such frame buffers are called floating point framebuffers. Tone Mapping uses simple Reinhard tone mapping algorithm.

3.9 Shadows:

I wanted to implement basic shadows because I think they are also a part of lighting calculations in computer graphics. A basic shadow mapping implementation using only single directional light as source and creating a depth texture from light's point of view and is captured and stored in a frame buffer colour attachment. This shadow depth texture is sent to lighting BRDF shaders and a simple checking is done if the current fragment is obscured by the fragment in depth texture. If yes, then current fragment is in shadow. To avoid shadow acne problem, a small bias is added to push shadow a bit below surface level. Also, a small PCF blur filter is applied to make shadows look smoother.

4 RESULTS AND EVALUATION

With all the implementations explained, the output results demonstrate all real time lighting effects showcasing different BRDFs and IBL along with post processing effects to make graphics look more aesthetically pleasing. The scene renders only one 3D object out of three which can be changed in runtime using ImGui. Image-based Lighting (IBL) also have worked perfected with can be changed as well at runtime. All the IBL captures are pre-computed at 1st frame. Total five HDR skyboxes are loaded.

The post processing effects can be individually enabled or disabled at runtime. Each effect comes with parameters can also be changed at runtime again using ImGui.

4.1 Profiling:

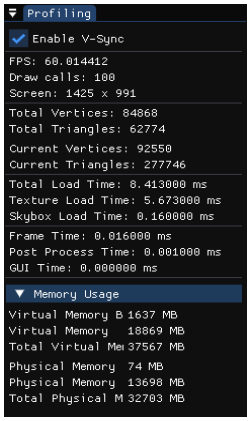

Figure 38: Profiling Details

A profiling manager class is created to record all the performance statistics. Again, Dear ImGui tool is used to record performance at runtime shown in Figure 38 above. Performance stayed at stable 60 FPS with V-Sync enabled and can be disabled which makes the FPS approximately 300-350 FPS. Draw calls is bit high because there is no instance rendering done. Below draw calls shown is total vertices and triangles of all the meshes loaded as

well as the count of vertices and triangles that is currently rendered in the scene.

The Total load time shows the loading time from start to end in milliseconds. Texture load time shows the time for loading all the textures for all meshes. Without using threads, the loading time was around 14-16 seconds which got reduced to 5-7 seconds using threads, but this also depends on the type of hardware the program runs on. The Skybox load time is the time of capturing and pre-computing all the HDR cubemaps for IBL which remained low. Frame time is the time between current frame and previous frame which is related to FPS. Post Process time is self-explanatory which is total time taken for rendering all post processing effects. Since, there were less effects hence, it didn't take that much time and lastly GUI time is time for ImGui to render which didn't show any delays.

The memory usage of program is shown, and it is interesting for me that the program uses more virtual memory from storage device than physical memory. This could be because textures files are big enough to be fit in RAM and OS decided to use storage device which in this case is an SSD as virtual memory here.

4.2 BRDF Comparison:

4.2.1 PBR Cook-Torrance vs PBR Disney:

There are not many dissimilarities between normal PBR and Disney BRDF in terms of looks. The internal working however of both BRDFs are slightly different. PBR Cook-Torrance one uses commonly used Lambert diffuse whereas Disney uses modified diffuse BRDF which is called as Burley Diffuse and it is mix of dielectric and metallic BRDFs which is dependent on metallic parameter. For normal distributions functions, Cook-Torrance uses the Schlick-GGX function which leaves quite big specular lobe and Disney used Trowbridge-Ritz for distribution as a result, specular lobe size is lot smaller. In terms of specular, Disney has better advantage in terms of control thanks to its additional parameters like anisotropic effect and clear coat, but with it requires more computations. The only similarity I found is both used same Fresnel-Schlick function for Fresnel effect.

4.2.2 Blinn-Phong Vs Oren-Nayar:

In terms of diffuse quality, Oren-Nayar here seems to be a winner as it is physically based, but also heavy. Blinn-Phong is good for lightweight and simple specular control, but it clearly violates energy conservation law as it reflects more light radiance than it receives. For specular, Oren-Nayar is also equipped with three different specular effects explained above which also gives an advantage of flexibility which Phong model doesn't have.

CONCLUSION

As hardware progressed fast, different graphical techniques also progressed a lot. The project's objective was to simulate real-time lighting techniques using different types of BRDFs and further enhance it using post processing. I mainly liked implementing different BRDFs like Disney and Oren-Nayar as well as Post Processing Effects like Bloom, SSAO and reducing the loading time of textures using threads.

Future Work:

There are few ideas I want to implement like better shadow mapping like Cascaded Shadow Maps (CSM), using Compute Shaders for Bloom and other heavy shaders, a proper material system that works on sub-meshes as well and the current project doesn't have correct draw order for transparent material and sub-meshes.

One idea for fixing transparency of sub-meshes is to extend sub-mesh structure such that each sub-mesh will have a bounding box and using the world space of bounding box to sort them by distance from camera and draw them from back to front.

Using multiple contexts of OpenGL for multi-threading and lastly, try out different graphics API like Vulkan, DirectX.

REFERENCES

[1] P. Shirley, "Basics of Physically-based Rendering". Nvidia, November 2012

[2] Bui Tuong Phong, Illumination for computer generated pictures, Communications of ACM 18 (1975), no. 6, 311-317.

[3] Warnock, J.E. A hidden-line algorithm for halftone picture representation. Dep. Of Comput. Sci., U. of Utah, TR 4-15, 1969

[4] Newell, ME.E, Newell, R.G., and Sancha, T.L. A new approach to the shaded picture problem. Proc. ACM 1973 Nat. Conf.

[5] Gouraud, H. Computer display of curved surfaces. Dep. Of Comput. Sci., U. of Utah, UTEC-CSc-71-113, June 1971. Also in IEEE Trans. C-20 (June 1971), 623-629.

[6] Neon Helium Productions, "Billboarding How To". [Online]. Available: https://archive.fo/SLzw2

[7] S. A. Coons, "Surface for computer-aided design of space forms", M.I.T., Cambridge, Mass, Project MAC. Tech. Rep. MAC-TR-41, June 1967.

[8] F. Ratliff, Mach Bands: Quantitative Studies on Neural Networks in the Retina. San Francisco: Holden-Day, 1965

[9] Joey De Vries, "PBR Theory". [Online]. Available: https://learnopengl.com/PBR/Theory

[10] Brian Karis, "Real Shading in Unreal Engine 4". Epic Games. [Online]. Available: s2013_pbs_epic_notes_v2.pdf (selfshadow.com)

[11] J, F. Blinn (1997). "Models of light reflection for computer synthesized. pictures". Proc. 4th Annual Conference on Computer Graphics: 192-198

[12] P. Beckmann and A. Spizzichino. "The Scattering of Electro-magnetic Waves from Rough Surfaces". Pergomon, New York, 1963.

[13] B. Walter, S. Marscher, H. Li, K. Torrance, "Microfacet Models for Refraction through Rough Surfaces". Cornell University, Beijing Institute of Technology, 2007.

[14] W. Matusik, H. Pfister, M. Brand, L. McMillan, "A Data-Driven Reflectance Model". TR2003-83, 2003. [Online]. Available: A Data-Driven Reflectance Model /Author=W. Matusik, H. Pfister, M. Brand, L. McMillan /CreationDate=July 28, 2003 /Subject=Computer Vision (merl.com)

[15] P. Hanrahan and W. Krueger. "Reflection from layered surfaces due to subsurface scattering". Computer Graphics Proceedings (SIGGRAPH 93), pages 165-174, 1993.

[16] M. Oren and S.K. Nayar, "Generalization of Reflectance Model". Columbia University, New York.

[17] M. Mittring, "Finding Next Gen – CryEngine 2". Crytek GmbH. Advanced Real-Time Rendering in 3D Graphics and Games Course – SIGGRAPH 2007.

[18] J. Jimenez, "Next Generation Post Processing in Call of Duty: Advanced Warfare". SIGGRAPH 2014.

[19] B. Burley, "Physically Based Shading at Disney". SIGGRAPH 2012. [Online]. Available: s2012_pbs_disney_brdf_notes_v3.pdf (dissneyanimation.com)