Advanced Graphics – Physically Based Rendering

Jainesh Pathak  
June 2023  
 MSc Computer Games Engineering  
Newcastle University  
 J.P.Pathak2@newcastle.ac.uk

ABSTRACT

A video game without lighting is never going to look pretty and will rather look uninteresting. Real world lighting calculation are complex and expensive for video games to handle. Many modern video games make use of a lighting calculation technique called Physically Based Rendering (PBR) which mimics the real-world lighting in a digital world and makes it look realistic. PBR works on microfacet surface model, energy conversations and Bidirectional Reflective Distribution Function (BRDF) to make it happen.

This report explains the process to achieve the approximation of realistic views using PBR and future for the project in context of video game graphics.

**KEYWORDS**

PBR; BRDF; Lighting; Lights; Irradiance; Billboards; HDR; Tone Mapping; Shading

**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 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 become more powerful; “close-to” realistic lighting also became possible.

**RELATED WORK**

1. *Lighting in old video games*

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.

* 1. *Light Diminishing*

Another fine addition to lighting Doom introduced in “Light Diminishing”. 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]

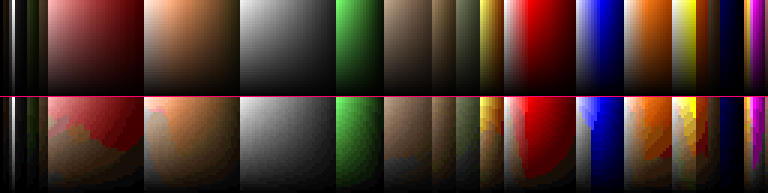
* 1. *Colormap implementation*

Before explaining Colormap, I need to first explain what color quantization is.

* + 1. *Color 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, color quantization works by reducing colors of the image such that the compressed is visually like its original.

In the PC port of original doom engine, it made use a Colormap which is a color quantized gradient texture which starts from a different color starting at top and slowly fades to black color as moving down. It is a precomputed lookup table which was used to fit in the game’s 256-color palette. The first 32 levels in the colormap were purely dedicated for implementing lighting. See Figure 1 for example.



**Figure 1: Top – Pure gradient. Bottom – Doom’s quantized colormap light levels**

Overall, the lighting calculations in Doom was simple and not realistic, but it did bring immersive gameplay which made the maps look and feel believable.

1. *Lighting in mid video games*

Games from Doom 3, Unreal Tournament onwards started introducing dynamic lighting and shadow casting. Different light sources were introduced like Point Light, Directional Light and Spotlight.

To quickly summarize these different light sources:

* 1. *Point Light:*

This type of light source emits lights in all directions. In technical details it contains the position in 3D world, color, intensity value and range.

* 1. *Directional Light:*

This light emits light 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. *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.

In real life, when a light falls onto an object, some of it get bounced or reflected and some get absorbed by the surface. The light which gets reflected leaves a white shining effect depending on the object’s shininess. This shining effect is called Specular highlights.

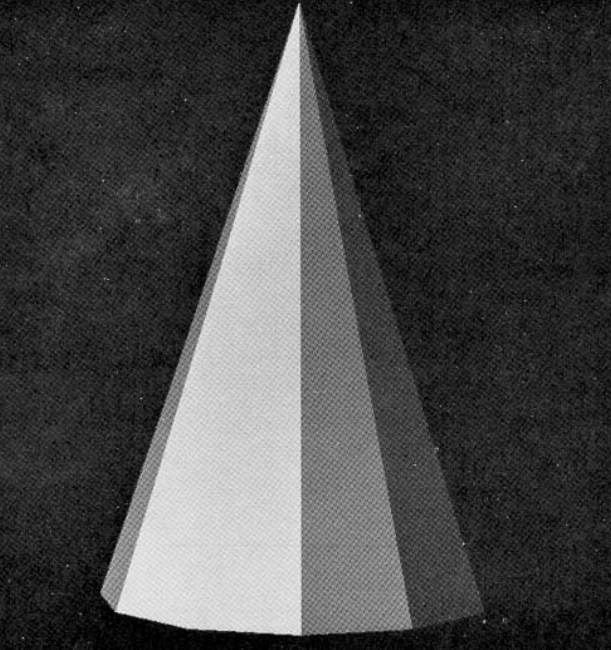
In computer graphics, the most popular shading model for specular highlights in the Phong shading model which he has explained in detail [2]. His work is influenced from previous shading model such that of Warnock’s [3] and Newell, Newell, Sancha’s shading model [4] and Gouraud shading model which needs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

* 1. *Warnock’s Shading:*

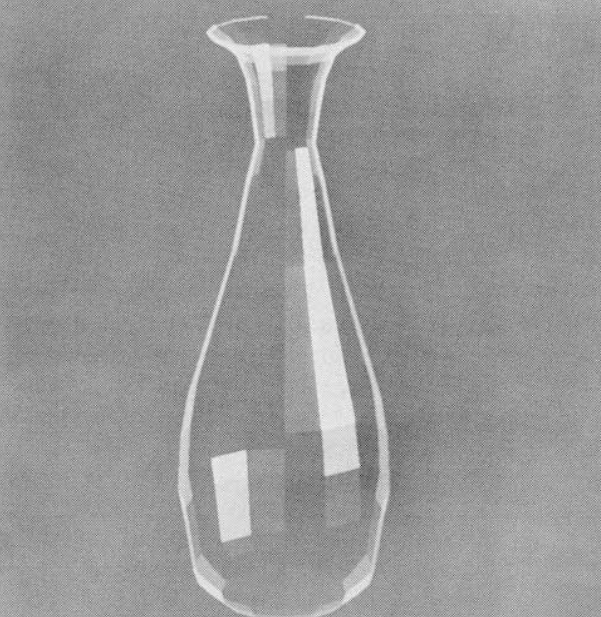
Light source and camera position are placed at the same position and function returns the sum of two terms: Normal and Specular reflection. Surfaces facing towards the light source are always bright compared to its adjacent which were shaded with different light intensities. Also, discontinuity can be easily seen on curved surfaces as seen in Figure 2.

* 1. *Newell, Newell, Sancha’s Shading:*

These people showcased an idea for creating specular highlights on transparent object. From experiments in real world, they found out that specular highlights are not created from a light source, but also from other objects as well which is highly depends on if the object has high reflective surfaces or transparent like glass. The application of this type of shading unfortunately is highly limited to transparent object and the specular highlights created on glass is like Warnock’s shading as it can be seen in Figure 3.



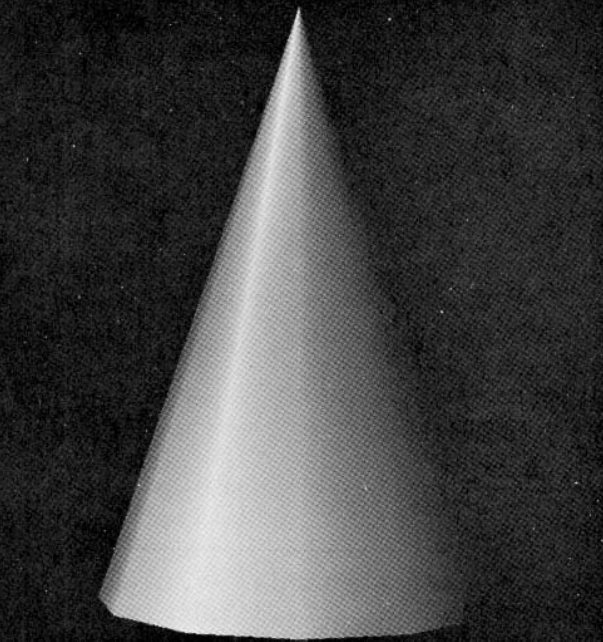
**Figure 2: Warnock’s Shading destroys smoothness on curved surfaces**



**Figure 3: Newell, Newell, Sancha’s Shading showing highlights on transparent object**

* 1. *Gouraud Shading:*

Gouraud [5] developed an algorithm which computes the specular highlights on curved surfaces of each vertex of the triangle. From the surface’s curvature, reflection intensity is calculated and is linearly interpolated towards zero between its edges connecting the neighbour surfaces. This generated visually appealing shading effect and retained the smoothness of curved surfaces as it can be seen in Figure 4.



**Figure 4: 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.

* 1. *Phong Shading:*

Bui Tuong Phong [6] introduced new shading model based on the Gouraud shading. It consists of three 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 color. 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 5: 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.

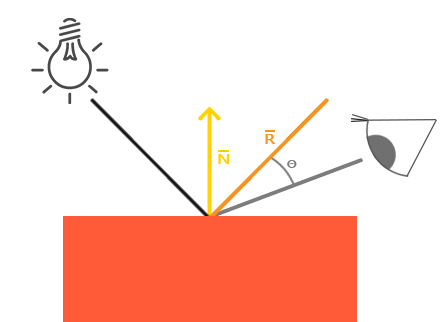
The normal vector is calculated from a triangle of a mesh. It is stored with the mesh data file as it is not required to calculate it every frame. To calculate the normal vector, we first take a difference vector from the two vertices to the one vertex and normalizing it and taking the cross product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

Once the angle is retrieved, it is simply multiplied with the light color and lastly added with the ambient component to create a believable image. This can be seen in Figure 6 with ambient lighting.



**Figure 6: Diffuse + Ambient Lighting on the mesh**

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



**Figure 7: 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 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 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 8.



**Figure 8: Phong Shading with shininess of 128**

**PROJECT APPROACH**

1. *Using OpenGL:*

After discussing various mentioned algorithms, for starters I have decided to continue to use the NCLGL framework provided in this master’s course and using OpenGL as the main graphics API.

The reason I chose OpenGL over other graphics API like DirectX and Vulcan is because it itself is a big graphics API with many features to learn and use it.

1. *Material System:*

Although the material system is not directly related to PBR. I want to create a 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 game engine like Unity, Unreal engine.

The current project directly makes use of shader and only one shader can be at runtime. With the help of the material system, this problem can be mitigated, such that materials can be changed at runtime simply by changing the index which will bind material and its shader at the current index and whatever other properties like color, textures are sent to the GPU.

1. *Post Processing:*

Post processing really interests me a lot because it gives the power to change the final image from one color to another. For this, I am planning to do only two post processing effects i.e., the Bloom and Screen Space Ambient Occlusion (SSAO).

* 1. *Bloom:*

For bloom, I plan to use a technique where the frame image only consisting of bright colours is first down scaled up to 8 times such that it looks blurry and rightly up scaled scale 8 times that the up scaled image looks closely like the original bright color image. This kind of Bloom effect was shown in Call of Duty: Advanced Warfare game. The bloom down scaling and up scaling will be done in the compute shader.

* 1. *Screen Space Ambient Occlusion (SSAO):*

I have also planned to do another post processing effect of Screen Space Ambient Occlusion (SSAO) for creating soft shadows around corners of the mesh.

**PROJECT PLAN**

I have extended the NCLGL library with abstract classes like the Frame Buffers, Texture, Uniform Buffer Objects. Making frame buffer objects will help me do post processing effects like Bloom as well as HDR with Tone Mapping and Image Based Lighting (IBL).

1. *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 just 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.

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

1. *Textures:*

Texture class was created to get flexibility on type 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.

1. *Tone Mapping:*

For Tone Mapping to work, the color 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. There are plans to use other tone mapping algorithms in the future.

1. *Billboarding:*

This is not related to lighting in the project. Billboard is a simple flat quad mesh object 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 [7], I will explain the type of billboards and its implementations. Billboards comes in three types:

* Point Sprites
* Axis Aligned
* Arbitrary Axis
  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.

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.

* 1. *Axis Aligned:*

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 was used in old games like Wolfenstein 3D and Doom.

* 1. *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.

* 1. *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:

[R1, U1, F1, Px]

[R2, U2, F2, Py]

[R3, U3, F3, Pz]

[0, 0, 0, 1]

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 just like normal.

ACKNOWLEDGMENTS

Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here. Insert paragraph text here.

REFERENCES

[1] Patricia S. Abril and Robert Plant, 2007. The patent holder's dilemma: Buy, sell, or troll? *Commun. ACM* 50, 1 (Jan, 2007), 36-44. DOI: <https://doi.org/>10.1145/1188913.1188915.

[2] Sten Andler. 1979. Predicate path expressions. In *Proceedings of the 6th. ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '79)*. ACM Press, New York, NY, 226-236. DOI:https://doi.org/10.1145/567752.567774

[3] Ian Editor (Ed.). 2007. *The title of book one* (1st. ed.). The name of the series one, Vol. 9. University of Chicago Press, Chicago. DOI:https://doi.org/10.1007/3-540-09237-4.

[4] David Kosiur. 2001. *Understanding Policy-Based Networking* (2nd. ed.). Wiley, New York, NY..

Conference Name:ACM Woodstock conference

Conference Short Name:WOODSTOCK’18

Conference Location:El Paso, Texas USA

ISBN:978-1-4503-0000-0/18/06

Year:2018

Date:June

Copyright Year:2018

Copyright Statement:rightsretained

DOI:10.1145/1234567890

RRH: F. Surname et al.

Price:$15.00