PBR and Image Based Lighting Renderer

May 5^{th}, 2023

A project by Vedang Parshuram Javdekar
NetID: VXJ210027

Summary

As a game developer, I am fascinated by the rendering techniques used in real time applications and different optimizations employed to achieve realistic results. PBR materials are one of such advancements, that closely follows the physical attributes to model materials that would produce realistic results. As opposed to statically rendered images that use *Ray Tracing* or *Path Tracing* techniques, real-time applications such as games constantly need to render the scene with a heavy time constraint (usually referred to as *'frame budget'*). During the classes, we have seen how Ray Tracers and Path Tracers calculate lighting to produce realistic results. However, such expensive tracing strategies cannot be employed in real-time. This suggests that even though PBR materials follow similar principles that Ray Tracers and Path Tracers use in terms of lighting but there are certain approximations and advanced techniques they require to correctly render the objects. I would be focusing on understanding those methods and lighting techniques that enable game engines to use PBR materials and produce high quality visuals.

PBR Image Based Lighting is a complex topic, and I would like to dedicate more time learning about it through this project.

Description of Work

Any real-time graphics application needs careful consideration in terms of processing, optimization, and software architecture to provide the best results possible.

The Framework

The project started with setting up a framework for the application that would abstract away certain things to allow reuse and flexibility in the systems. The application takes inspiration from game engines in terms of system and API design. An excellent book 'Game Engine Architecture' by Jason Gregory, illustrates a few solutions related to engine startup and cleanup, which were found useful and hence were implemented in the project.

```
// Initialization
m_IsRunning = true;
for (auto& system : m_SubSystems)
{
    if (!system->Init(settings))
    {
        m_IsRunning = false;
        break;
    }
}
```

```
// Shutdown
for (auto itr = m_SubSystems.rbegin(),
        end = m_SubSystems.rend(); itr != end; ++itr)
{
    (*itr)->Shutdown();
}
m_SubSystems.clear();
```

This provided benefits such as having full control on the order in which systems get initialized and shutdown. The project relies on a handful of third party libraries and setting those up early in the project paid out well.

Dependencies

The following libraries are used in the project:

Library	Usage
GLFW	Windowing and Inputs
GLAD	Provide an interface to call OpenGL functions
glm	The recommended OpenGL Math library
assimp	Import and Export models in multitude of formats
imgui	Immediate Mode GUI library to provide on screen controls

Setting these up the project in CMake and making all these libraries work with the existing framework was a quite a bit of challenge. But once everything was setup, it was very convenient to use it.

Windowing and Input

With a few custom classes and callback functions `glfw` provides, **Keyboard, and Mouse** interactions were in place. More details could be found in the respective subsystems in code.

OpenGL Setup

OpenGL has many moving parts and even to render a single triangle, it takes a fair amount of work.

The following image shows the rendering pipeline for OpenGL:

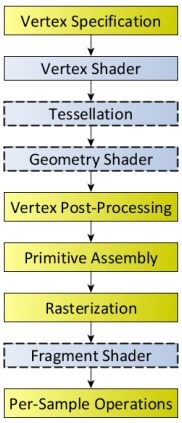


Figure 1: RenderingPipeline

The rendering pipeline first gets the vertex specification (vertex data and layout), then assembles the vertices, pass them through the vertex shader that usually transforms the vertices with the relevant Model, View and Projection matrices. Then all the fragments that would be shown on the screen go through the fragment shader.

This whole process is a bit tricky and there are many OpenGL specific concepts in the play. This was an introduction to Vertex Arrays, Vertex Buffers, Element/Index Buffers and how they help OpenGL not only reduce the memory usage by avoiding duplication, but also the memory for the object could be reused.

The **Vertex Array** has **Vertex Attribute Pointers** and they store how the data is laid out in the **Vertex Buffer**. The **Vertex Buffer** stores the actual data, i.e. vertex position, normal, tangent, texture coordinates,

etc. Usually its a flat array of **floats**. The vertex attribute pointers and stride and offset let the GPU know how to represent that flat array of floats.

Rendering with vertices sometimes would require duplication of data to maintain the correct mesh properties(such as flat shading). Hence, **Index Buffers** are used that store the indices of the vertices that make up the faces for an object. This avoids repetition of the data in the vertex buffer.

A classic exmple given to illustrate this is how to render a quad. A quad consists of 4 vertices, and it can be drawn as 2 triangles that share a side.

if we were to draw that quad without indices, it requires the 2 vertices to be duplicated.

```
float vertices[] = {
    // first triangle
     0.5f,  0.5f, 0.0f,  // top right
     0.5f, -0.5f, 0.0f,  // bottom right
    -0.5f,  0.5f, 0.0f,  // top left
    // second triangle
     0.5f, -0.5f, 0.0f,  // bottom right
    -0.5f, -0.5f, 0.0f,  // bottom left
    -0.5f,  0.5f, 0.0f   // top left
};
```

And if we instead use index buffer, only 4 vertices are required. This is not always the case though. For flat shading of a model with only quads, we still need to have 4 duplicated vertices each with a different normal vector.

```
float vertices[] = {
     0.5f,  0.5f, 0.0f,  // top right
     0.5f, -0.5f, 0.0f,  // bottom right
    -0.5f, -0.5f, 0.0f,  // bottom left
    -0.5f,  0.5f, 0.0f   // top left
};
unsigned int indices[] = {  // note that we start from 0!
    0, 1, 3,   // first triangle
    1, 2, 3    // second triangle
};
```

It might not look like a huge improvement when there are small number of vertices, but it becomes significant when the input size increases to a few thousand triangles.

So with all the setup done correctly, the final call to render an object looks as follows in the framework that has been built for this project.

```
void Renderer::Render(Mesh* mesh, Shader* shader)
{
    shader->Bind();
    VertexArray* va = mesh->GetVAO();
    if (!va) { return; }
    va->Bind();
    IndexBuffer* ib = mesh->GetIBO();
    if (!ib) { return; }
    ib->Bind();

    glDrawElements(mesh->GetRenderMode(), ib->GetCount(), GL_UNSIGNED_INT, 0);
}
```

The render mode could take different values depending on the indices, e.g. for meshes imported via *assimp*

use `GL_TRIANGLES` while the primitives that have been generated in code use `GL_TRIANGLE_STRIP` as triangle strip saves some space on index buffers.

Even after all this setup the object was rendering in the **Normalized Device Coordinates**. That means the width and the height of the window was mapped to a **[-1.0, 1.0]** range.

Shaders were such a crucial part of the project that required more work and attention than anything else. A basic version of `Shader` class was written initially and as required more features were added.

Shaders

Mainly the vertex and fragment shaders were required in this project for correct rendering and calculations. The shaders are small programs that run on GPU at different stages in the rendering pipeline. They too need to be compiled and linked before they can be used("bound"). The shaders can be injected with external data via either *vertex attributes* or *uniforms*. Vertex attributes are passed from the bound vertex array object. But we can set uniforms afterwards and that would upload the new values to GPU.

Shader Preprocessing

During the development of this project, I implemented this feature that allows me to easily access shader uniforms dynamically in the ImGui. This particularly helps when there are uniforms that need to be manually set.

The way this system works is, it searches for the word uniform in the shader code. If it is found then the code tries to parse the data type and the uniform name. Once all of this data is acquired, these uniforms are added to a special set, so that they can't be set from the code, and would need explicit setting through the UI. According to my needs, I added the following features:

1. Ignore attribute:

```
// User need not have explicit control over this uniform
// And it can be set via code.
/*Ignore*/ uniform vec3 uViewPos;
```

2. Color attribute:

```
// Will be shown as a ColorEdit3 in the UI as opposed to a drag float 3.
/*Color*/ uniform vec3 uAlbedo;
```

Currently the system supports the following types:

```
enum class UniformType :uint8_t
{
    NONE = 0,
    FLOAT,
    BOOLEAN,
    FLOAT2,
    FLOAT3,
    COLOR3,
    FLOAT4,
    COLOR4
};
```

This is responsible for generating the fields in the Shader Uniforms panel in the UI.

Camera

Project uses a very flexible and extensible camera system. The current camera used in the Demo is just an extention that puts some constraints on the camera motion to get orbiting camera.

Events are processed to react to inputs for camera, such as right click and drag moves the camera according to the mouse motion, scroll wheel is used for zooming and other such features are included in the current camera system.

Additionally, for precise control, camera draws in the ImGui panel too. This allows easy manipulation of values in realtime and the user need not close the application to change the values. This improves user experience as well as helps a ton with the development times.

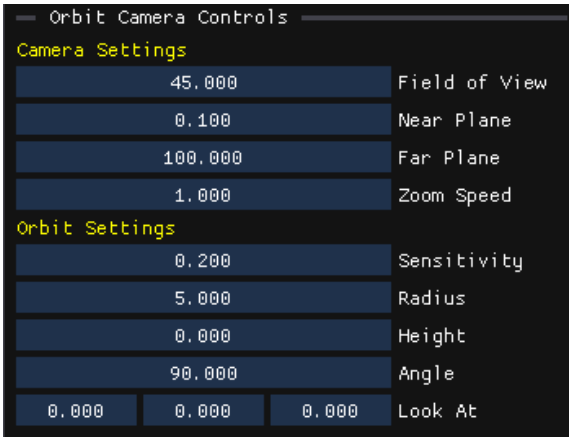


Figure 2: Camera Settings In ImGUI

The keyboard and mouse controls force the camera to recalculate view and projection matrices. Changing any value in this interface also forces the view or projection matrices to be recomputed. An appropriate shader then reads these matrix values every frame and uses them in vertex shader to compute the world position of a vertex.

Textures

Textures are loaded using the `stb_image.h` header only library. These textures can be manually created with color data as well. To use textures, we need to bind the texture in a texture slot on the GPU and let `sampler2D` in the shader code know where to sample the texture from.

Framebuffers and Gamma Correction

The gamma correction is required to fix the color space differences in the calculations and the colors in the textures. So, we apply gamma correction as a post process effect. Along with depth test and stencil test, the whole scene is renderered to a framebuffer. And then a gamma correcting fragment shader corrects the gamma for the whole scene. This way it is more convenient than correcting in every object shader.

PBR Theory

Physically based rendering technique was pioneered by researchers at Epic Games. For realistic results it makes use of the microfacet model that was seen in the class.

During the class, we looked at render equation that recursively calculates the output radiance based on the light sources and material properties represented in form of BRDF. Physically based rendering uses a special version of render equation called reflectance equation:

$$L_0(p, \omega_0) = \int_\Omega f_r(p, \omega_i, \omega_o) \cdot L_i(p, \omega_i) \cdot (n \cdot \omega_i) d\omega_i$$

To make it computationally possible, we discretize it and convert the integral into a Riemann sum. As the fragment shader goes over every fragment, computing the radiance at each fragment will give us the accurate lighting.

BRDF

The physically based rendering is based on the following principles:

- Mircofacet model
- Energy Conservation
- Physically based BRDF

Blinn-Phong that was used for lighting in the first implementation, doesn't adhere to the Energy conservation principle that states that the outgoing light energy can never exceed the incoming light energy with an exception of emissive materials. Hence we cannot use Blinn-Phong for physically based rendering.

We use something called as a **Cook-Torrance BRDF**. Almost all physically based renderers use it.

It consists of 2 parts:

1. The energy that is reflected (specular term)
2. The energy that is refracted (diffuse term)

Mathematically,

$$f_r = k_d \cdot f_{lambert} + k_s \cdot f_{cook-torrance}$$

The first term is the Lambertian Diffuse term that is given by:

$$f_{lambert} = \frac{c}{\pi}$$

where c is the surface albedo or surface color.

The specular part looks like:

$$f_{Cook-Torrance} = \frac{DFG}{4(\omega_0 \cdot n)(\omega_i \cdot n)}$$

The three functions are described as follows:

1. Normal Distribution Function **(D)**: approximates the amount the surface's microfacets are aligned to the halfway vector. This can be controlled by controlling the roughness parameter of the surface. This function is responsible for modelling the microfacet as accurately as possible.
2. Fresnel Function **(F)** The Fresnel equation describes the ratio of surface reflection at different surface angles.
3. Geometry Function **(G)**: describes the self-shadowing property of the microfacets. When a surface is relatively rough, the surface's microfacets can overshadow other microfacets reducing the light the surface reflects.

There are more than one approximations available for these functions for calculating the specular lighting, but some very smart folks at Epic Games have been researching over this topic and finally came up with a workflow/approximations that would give realistic looking renders in realtime.

From the resources I came across, I would be using **Trowbridge-Reitz GGX** for distribution function, **Smith's method and Schlick's GGX** for geometry and **Fresnel-Schlick** approximation for frenel.

Normal Distribution:

$$NDF_{GGXTR}(n, h, \alpha) = \frac{\alpha^2}{\pi((n \cdot h)(\alpha^2 - 1) + 1)^2}$$

Geometry:

$$G_{SchlickGGX}(n, v, k) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k}$$

Smith's Method:

$$G(n, v, l, k) = G_{SchlickGGX}(n, v, k) G_{SchlickGGX}(n, l, k);$$

The two terms are there for geometry obstruction and geometry shadowing.

Fresnel:

$$F_{Schlick}(h, v, F0) = F0 + (1 - F0)(1 - (h \cdot v))^5$$

It has been determined that a value of $F0 = 0.04$ works well for materials and we mix it with albedo to get intermediate results.

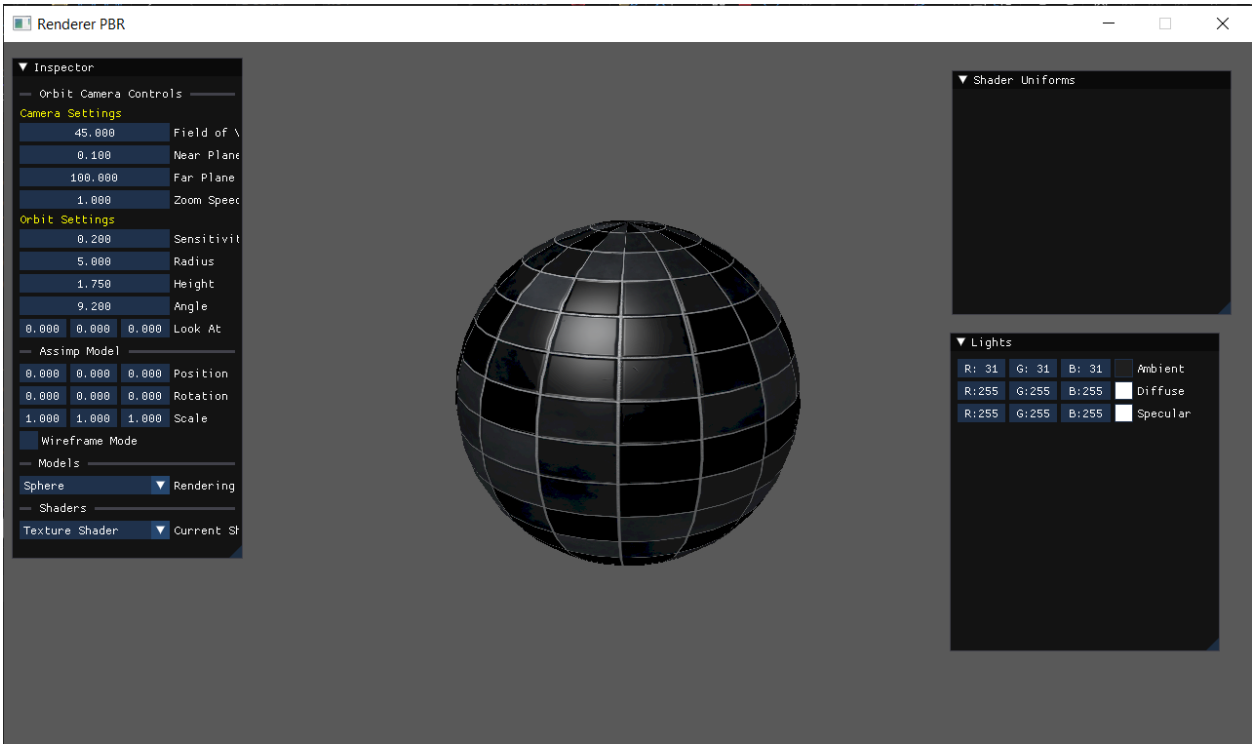
With all these functions it was straight forward implementation for direct lighting for PBR.

All the three functions were implemented in the Fragment Shader and Normal Mapping was used from the previous Blinn-Phong lighting shader.

Results

Texture Shader(Blinn Phong lighting):

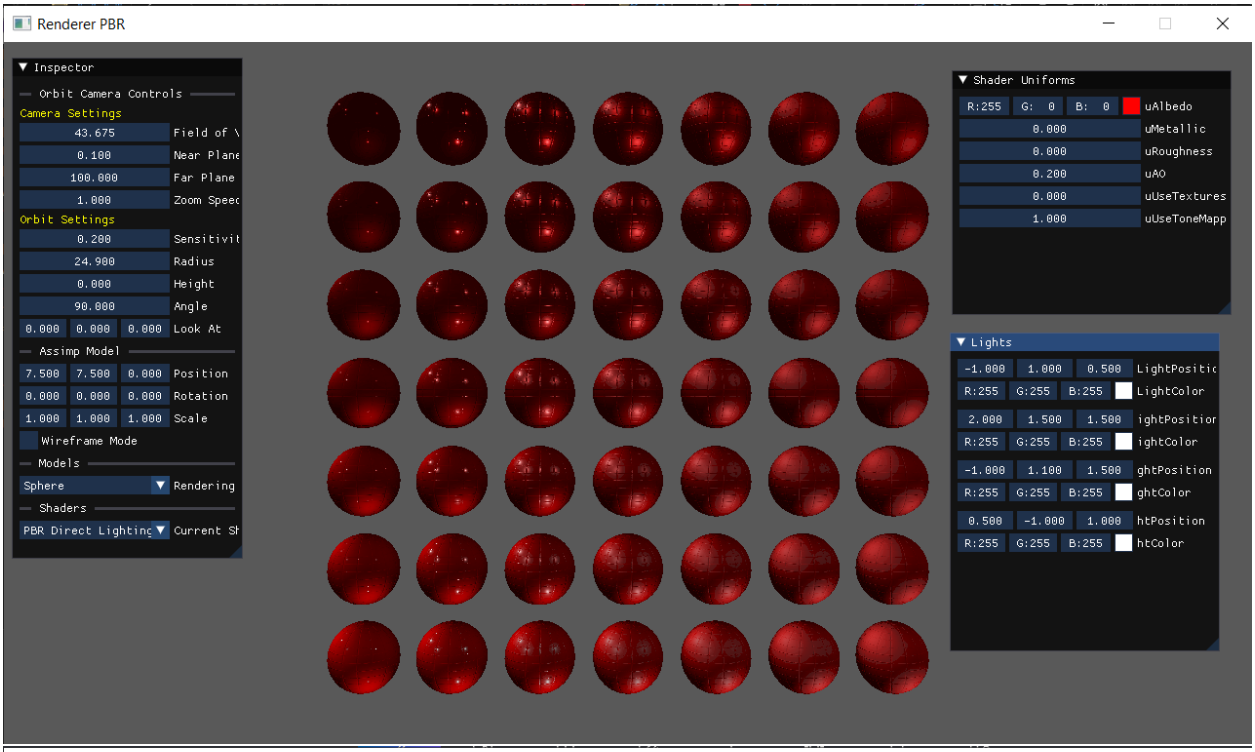
This is the output of the normal mapped model with Blinn Phong lighting:

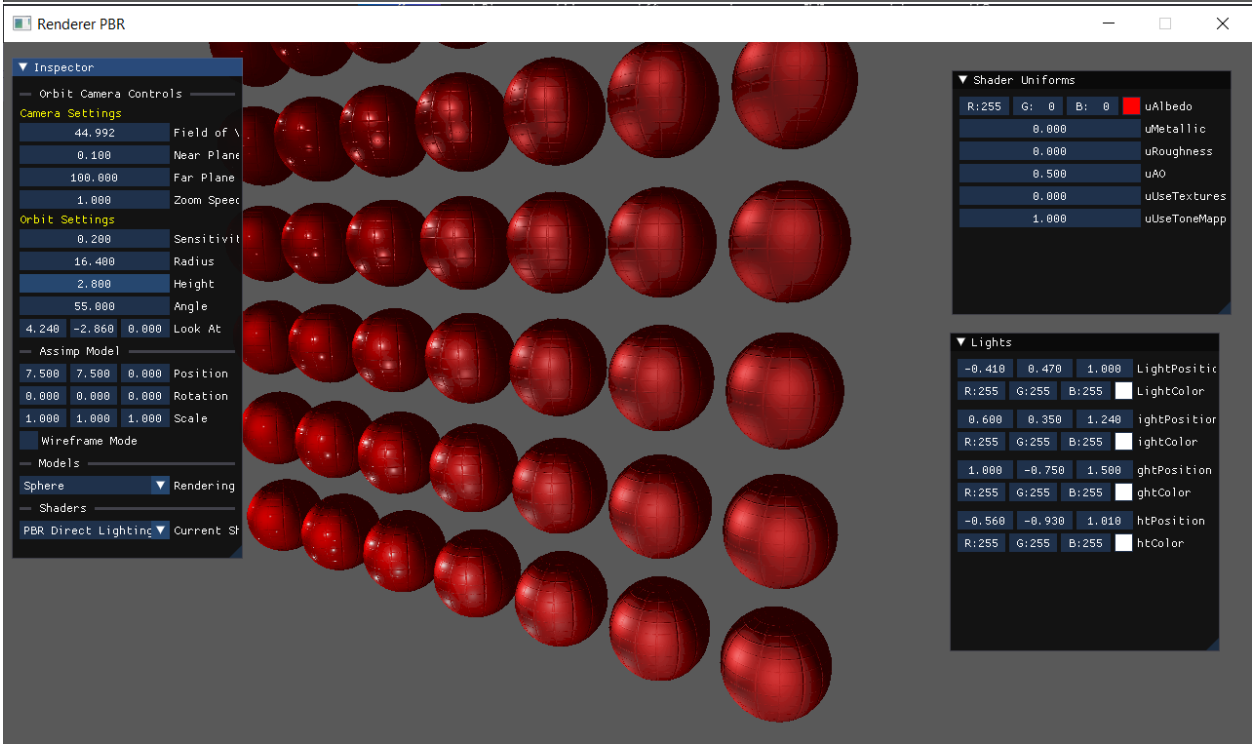


As this passes through the Framebuffer, gamma correction is applied to rendered image.

PBR Direct Lighting Shader(Cook Torrance BRDF):

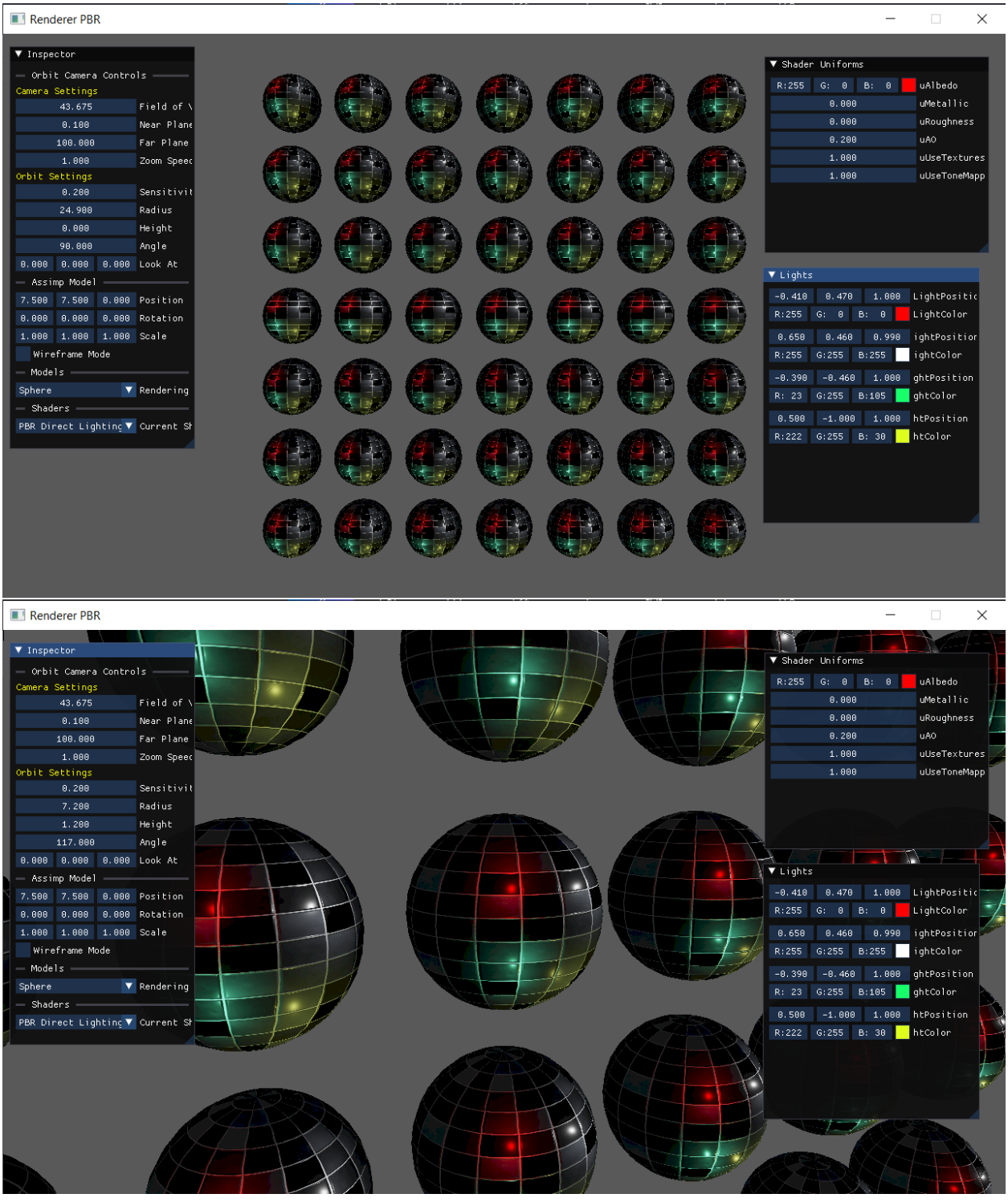
This is an output of the PBR direct lighting shader that illuminates the scene with 4 lights and each of the sphere has increasing metallic and rougness values.





PBR Textured Direct Lighting:

Now instead of fixed values throughout, we are sampling from different textures.



It can be seen that because we are using textures, each value produces a different reaction to light by that portion of the mesh.

Analysis of Work

The project is currently at the stage where it could load an arbitrary model, and with given textures, it could render through the PBR direct lighting technique. Initially, doing Image based Lighting was in the project roadmap, but implementing IBL is not only theoretically complex to comprehend, but it is equally challenging to implement given the additional work that needs to be done to create irradiance maps and sample those irradiance maps to create a plausible outputs.

With current setup the materials look decent, not very realistic, but a lot better than simple blinn-phong. Irradiance maps and image based lighting will only have additional environmental lighting.

Overall, the project met all the other goals except for the image based lighting. The application allows the user to change paramaters in realtime and see the results.

Working on this project was ground work for something such as writing your own game engine. Working with OpenGL was a bit of a pain due to inconsistent API, but it was the only API that was easy to pick up and that required less boilerplate code.

It's an intentional choice of not using any framework such as bgfx or the like that would provide me with the renderer, as that would not have given me an opportunity to implement all these things on my own and have a better insight into graphics programming. With a renderer my contribution would then be just writing the PBR shaders and creating an application in that framework which would not have been valueable to me. I got an opportunity to study concepts such as HDR, tone mapping and framebuffers, while working on the project.

Based on my current understanding I find Image based lighting approachable but given the time constraints and my framework lacking the necessary abstractions to handle cubemaps and their sampling, I would be submitting the project in the current state.

References

- For all the OpenGL and PBR concepts and mathematics: https://learnopengl.com/
- For creating a custom primitive sphere mesh: http://www.songho.ca/opengl/gl_sphere.html
- Many Videos from this playlist to abstract OpenGL functionality into classes: https://youtube.com/playlist?list=PLlrATfBNZ98foTJPJ_Ev03o2oq3-GGOS2