Advanced Computer Graphics

Final Project

Colin Orian Student Number: 100622430

December 13, 2019

How to Run and Control

Open the Solution file and run the program.

- W,A,S,D: move the camera
- 1: Toggle HDR
- 2: Toggle Bloom
- 3: Toggle Roughness
- 4: Toggle Metalic

Goal of the Project

In our classes we talked about several different rendering algorithms. However, throughout my experience in computer graphics I have mainly used Phong or Bling-Phong rendering. For this project, I wanted to experiment with more complex real-time rendering algorithms and explore different techniques to enhance a rendered scene.

Choosing the Rendering Algorithm

I didn't want to implement ray-tracing for this project. Ray-tracing takes a significant amount of time so it wouldn't be ideal for real-time rendering. My undergraduate thesis also involves ray-tracing and I want to expand my breadth of knowledge instead of focusing more on ray-tracing. While I was exploring different rendering techniques I learned about High Dynamic Range (HDR) rendering. I then implement Physical Based Rendering (PBR) because I wasn't satisfied with the results of HDR.

Implmenting High Dynamic Range

Colours in the real world have a very wide range. However, colours on a computer screen are bounded from 0.0 to 1.0 [1]. Due to this limitation, colours on a computer screen may not portary the real world realistically [1]. For example, a light bulb's light intensity is significantly less than the Sun's [1]. Without HDR, a lightbulb's light and the Sun's light may have the same intensity (1.0f, 1.0f, 1.0f).

To simulate the real spectrum of colour, we need to first render the scene to a frame buffer texture so OpenGL doesn't automatically bind the colours from 0.0 to 1.0 [1].

```
glGenFramebuffers(1, & hdrBuff);
glBindFramebuffer(GL_FRAMEBUFFER, hdrBuff);
createFramebufferTexture(& hdrColor, WIDTH, HEIGHT, GL_RGBA32F, GL_COLOR_ATTACHMENT0);
```

We then render a plane to the scene and texture the plane with previous rendered screen[1]. Once we do that, we can then apply a tone mapping algorithm to convert the colours back into the range of 0.0 to 1.0[1].
Although I implemented HDR, I felt like my project was missing a large amount of work. Furthermore, the rendering algorithm I used to render the scene was still Phong rendering and I wanted to learn a new rendering algorithm instead of apply post processing to Phong rendering.

Implmenting Physical Based Rendering

PBR was exactly what I was wanting to achieve with this project. Unlike Phong rendering, PBR takes the microsurfaces of a material into consideration when rendering[2]. The algorithm that I found also implements a more complex

BRDF compared to Phong shading[2]:

$$brdf_{cook-torrence} = \frac{D * F * G}{4 * (w_o \cdot n) * (w_i \cdot n)} \tag{1}$$

where

$$D = \frac{\alpha^2}{\pi * ((n \cdot H)^2 * (\alpha^2 - 1) + 1)} \tag{2}$$

$$G = (\frac{n \cdot w_i}{(n \cdot w_i) * (1 - k) + k}) * (\frac{n \cdot w_o}{(n \cdot w_o) * (1 - k) + k}) \tag{3}$$

$$F = F_0 + (1 - F_0) * (1 - (H \cdot w_o))^5 \tag{4}$$

$$w_i = lightToPoint$$

$$w_o = viewToPoint$$

$$N = normal$$

$$\alpha = k = roughness$$

$$F_0 = indexOfRefraction$$

$$indexOfRefraction = mix(baseReflect, albedo, metalic)$$

$$H = HalfwayVector$$

The albdeo, metalic, baseReflect and roughness values usually come from material maps created by material artists[2]. By utilizing the Cook-Torrence BRDF, we can find how much a light contributes to the colour at a given pixel[2].
For each Light Source:

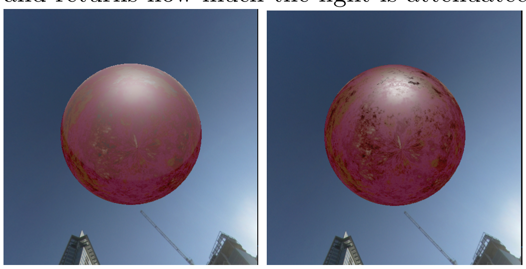
$$colour+ = refractFactor * \frac{albedo}{\pi} + brdf_{cook-torrence} * radiance * (w_i \cdot n) \tag{5}$$

where

$$radiance = lightColour * attenuation()$$

$$refractFactor = (1 - F) * (1 - metalic)$$

The attenuation() function takes in the distance from the position to the light and returns how much the light is attenuated due to travelling.



The above pictures is the fully implemented PBR. The left image uses a constant value for roughness while the right image uses a roughness map. The roughness map, metalic map and texture map were created by Your Perfect Studio.

Miscellaneous

Before I found PBR, I implemented bloom and sun shafts. Bloom was easy to implement because I had already implemented HDR. To implement bloom, I made another frame buffer texture and only drew the pixel's colour to it if the pixel's colour was above a certian value[3].

```
layout (location = 1) out vec4 bright_color;
...
bright_color = vec4(0.0f);
if(result.x > 1.3f || result.y > 1.3f || result.z > 1.3f){
bright_color = vec4(result,1.0f);
}
```

Once I made the bloom texture, I applied a Gaussian Blur to the it and added it to the plane's texture[3].

The above picture is the bloom texture of a scene with the Stanford Dragon. The majority of it is black because most of the pixels are less than 1.3f

I implemented sunshafts by, once again, using a frame buffer texture. I gave each object in the scene an isEmitter variable. If the object is an emitter, the texture would be coloured white. Otherwise, it would be coloured black. This gave me an occlusion map[4]. For each pixel, I sent a ray from the pixel location to the light sources. I then sampled the occlusion map on this ray. As the sample got further away from the original pixel, the intensity of the sampling decreased[4]. I summed all the samples together and added the result to the output colour.

References

[1] HDR. (n.d.). Retrieved December 13, 2019, from https://learnopengl.com/Advanced-Lighting/HDR.

[2] Theory. (n.d.). Retrieved December 13, 2019, from https://learnopengl.com/PBR/Theory

[3] Bloom. (n.d.). Retrieved December 13, 2019, from https://learnopengl.com/Advanced-Lighting/Bloom

[4] Mitchell, K. (2007). Volumetric Light Scattering as a Post-Process. *GPU*

Gems (Chapter 13). Boston, MA: Pearson Education, Inc. Retrieved December 13, 2019, from https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch13.html