Physically Based Rendering

Documentation

Iker Galardi

Revision	Date	Changes
1	26/8/2021	Initial document
2	27/8/2021	Corrections of language

Contents

1 Introduction

PBR-Renderer is a project that implements Physically Based Rendering (or PBR for short) using the Cook-Torrance model.

2 PBR and Cook-Torrance model

PBR is a collection of rendering techniques and models that allow the creation of more "realistic" graphics for the end user. As the name implies, physically based rendering takes reality as a base to model how light interacts with objects. These models are also called Bidirectional Reflectance Distribution Functions (or BRDF for short) and express mathematically how light is reflected or absorved on materials.

Physically based rendering techniques, to be considered physically based, need to follow the next two rules:

- Use a microfacet surface model.
- Energy conservancy.

There are many models that meet the before mentioned requirements, but the most used one on realtime rendering systems is *Cook-Torrance*, and its what was selected for this renderer as well. Other more modern

This model follows the microfacet theory. The theory describes that any surface at the microscopic scale can be described by tiny little perfectly reflective mirrors, called microfacets. The more rough the surface is, the more disorganized those microfactes, as it can bee seen on figure 1.

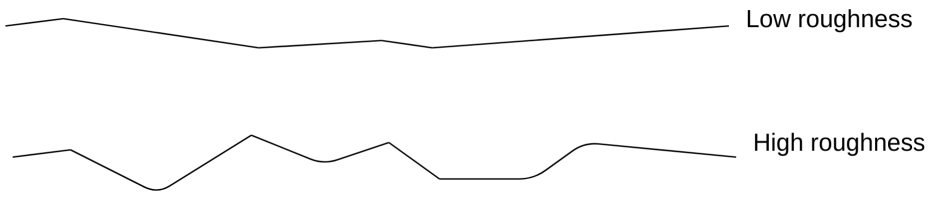

Figure 1: Roughness representation at the microscopic level

Lighting is divided into two parts: diffuse (refracted light) and specular (reflected light). All the light hitting the material needs to be refracted or reflected, thus, with the next equation we express that relation:

$$k_s + k_d = 1$$

Knowing that, the final light calculation is done the next way:

$$f = k_s \cdot f_{specular} + k_d \cdot f_{diffuse}$$

The model used for the diffuse part is known as the *Lambertian* model:

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

The *Cook-Torrance* model takes into account the before mentioned microfacets, and encodes the light interaction on the next equations:

$$f_{specular} = \frac{N \cdot G \cdot F}{4 \cdot (w_0 \cdot n) \cdot (w_i \cdot n)}$$

$$N = \frac{a^2}{\pi \cdot ((n \cdot h)^2 \cdot (a^2 - 1) + 1)}$$

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

$$F = F_0 + (1 - F_0) \cdot (1 - (h \cdot v))^5$$

It is important to note that these model only supports dielectric materials (non metallic), as metallic surfaces react differently to light.

3 Renderer architecture

The renderer implements a basic forward rendering technique where the model's final pixel is calculated every time a mesh is rendered. This technique has some drawbacks, which are related to multilight scenes. This drawback doesn't affect this renderer, as all the light comes from a single directional light. Thus, this technique being simpler to implement is perfect for the purpose of this project.

3.1 OpenGL abstraction

To simplify code, all *OpenGL* object are abstracted away into classes. This would also be useful when adding more graphics APIs.

The next are the available objects:

- **Buffer**: this object represents both vertex and element buffers.
- **VertexArray**: represent *OpenGL* vertex array object.
- **Shader**: represent an *OpenGL* shaders and give some utility functions to work with them.
- **Texture**: represents an *OpenGL* texture and set some default settings.

The before mentioned classes are just simple wrappers that should not add much overhead.

Some interesting parts of the abstraction are shaders. Normally shaders are divided into two files, for vertex and fragment parts; this is a pain to work with, so a little preprocessor has been written to have a single shader file and internally divide them.

```
#shader vert
[...]

#shader frag
[...]
```

As seen on the example code, the shaders are divided with preprocessor `#shader`, and fed into *OpenGL* as two different strings, making it totally transparent from *OpenGL*'s point of view.

3.2 Material system

The renderer does not have any advanced material system, as the only parameters passed to the shaders are the roughness, normal and color textures. Thus, the *Model* simply stores those textures and the renderer binds them when rendering the object.

3.3 Normal mapping

Normal maps come in what is called *tangent space*. Normally the most components for the transformation matrix to worldspace is precalculated, but instead on this engine calculates it at runtime on the fragment shader using the next code:

```
vec3 calculate_normal_in_world_space(vec3 normal_texture) {
    vec3 texture_normalized = normal_texture * 2.0 - 1.0;

    vec3 Q1  = dFdx(v_fragment_position);
```

```
        vec3 Q2  = dFdy(v_fragment_position);
        vec2 st1 = dFdx(v_texture_coordinates);
        vec2 st2 = dFdy(v_texture_coordinates);

        vec3 N   = normalize(v_normal);
        vec3 T  = normalize(Q1 * st2.t - Q2 * st1.t);
        vec3 B  = -normalize(cross(N, T));
        mat3 tbn = mat3(T, B, N);

        return tbn * texture_normalized;
    }
```

3.4 Vertex shader calculations

Vertex shader is pretty simple, as it only transforms the vertices into worldspace, and then into clip space to be rendered. Matrices to transform the object are passed through uniform variables. The transformation is done with the next line of code:

```
gl_Position = u_proj * u_view * u_model * vec4(in_position, 1.0);
```

Appart from that, this shader also prepares some variables to pass to the fragment shader, such as the fragment position in world space (`v_fragment_position`), normal (`v_normal`), texture coordinates (`v_texture_coordinates`). All the passed values interpolated to calculate the value they should take for the specific fragment automatically by *OpenGL*.

3.5 Fragment shader calculations

The fragment shader is where all the PBR stuff resides. If the previous *Cook-Torrance* theory is clear, this shader is pretty straight forward. The most important part is how the roughness gets remapped into the variable α and k. Both of the variables are remapped like *Unreal Engine* does:

$$\alpha = roughness^2$$

$$k = \frac{(\alpha + 1)^2}{8}$$

Knowing that, the shader basically translates the mathematical expressions to GLSL.

```
float numerator = normal_distribution * fresnel_distribution * geometry_distribution;
float denominator = 4 * max(dot(N, V), 0.0) * max(dot(N, L), 0.0) + 0.001;
float cook_torrance_specular = numerator / denominator;
```

It is important to note that addition, as it prevents any divitions by 0 and weird errors that can produce.

3.6 Antialiasing

The engine enables MSAA (MultiSampling Anti Aliasing) via *OpenGL* and *SDL*. This multisampling technique calculates multimple subpixels and averages them to create the final pixel.

4 Further improvements

The renderer still is very bare, as it only implements the PBR shader and all the necessary CPU side stuff. The next stuff is necessary to have a much better base:

- **Material system**: having a material system at all would be nice. The material system should handle the necessary textures and shaders.
- **Scene system**: having scenes organized in a proper way would help reduce client side code. This would mean that all the mesh/model abstractions need to be redone. A renderable class with materials and meshes would help to organize better the code.
- **Better graphics abstraction**: the currently implemented abstraction is very simple, as it's just a simple wrapper for certain objects. Classes currently don't do deep copies, and that lead to bad behaviour. Also, most of the functionality is not supported. Apart from that, as it's deeply tied with *OpenGL*, adding other APIs would be difficult.