The Quarantamaula Engine

Sergio 'Pepis' *Fabregat*

May 24, 2025

Figure 1: Two enemy checkers making peace to enjoy a pretty sunset on top of a chess board. Image rendered with Quarantamaula.

1 Introduction

Quarantamaula is a lightweight graphics engine based on OpenGL 4.6. Its purpose was to teach computer graphics and rendering techniques to its developer, who at the time hadn't had any formal education on the subject. As such, its feature set is minimal, and focused exclusively on graphics. There's no collision detection, scene graph, or large-scale optimizations. What it *is* good at is showcasing models in small scenes with eye-catching lighting.

Throughout this document we will delve deeper into its most remarkable features, which are:

- **Deferred Shading**
- **PBR**
- **Shadows**
- **IBL**
- **Model Loader**

2 Deferred Shading

Deferred shading or deferred rendering is a technique employed to improve performance by only performing lighting calculations on fragments that will actually end up on screen. Normally, in what is called forward rendering, fragments are colored as they come. The problem with this is that you might compute really complex and thorough lighting for a fragment, and then the fragment that comes next is in front of the previous one. Which means it will end up on screen, while the one you just shaded will not.

In order to avoid this, we divide rendering into two stages, the geometry pass and the lighting pass. During the geometry pass we draw all objects as we normally would, but the fragment shader runs are employed to write into a collection of textures called the G Buffer all the information we need to compute lighting. That is, position, albedo, normals, etc.

Then, for the lighting pass, in those textures written during the geometry pass we will have the information needed to compute the color of only the pixels that will end up on screen. So we take the data, perform our calculations, and paste the result on top of a quad that covers the whole screen.

2.1 Geometry Pass

As explained earlier, Quarantamaula loops through a list of GameObjects, rendering each and in the process writing all the data needed for lighting. The G buffer is composed of 5 textures, all of the 16 bit float RGBA format:

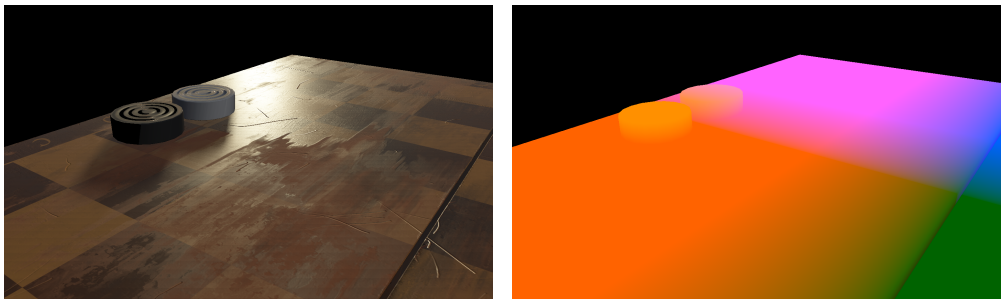

Figure 2: On the left the final image, on the right the contents of the of the position G buffer.

- **Texture 1:** The RGB components store the position (Figure 2), while the fourth stores roughness.
- **Texture 2:** The RGB components store the normals (Figure 3), while the fourth stores the specular parameter.

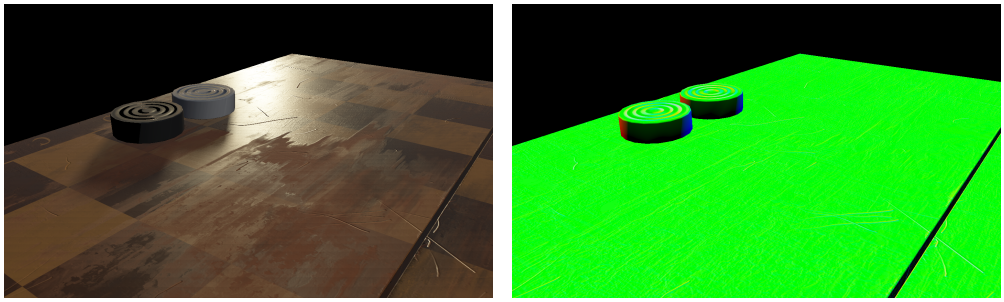

Figure 3: On the left the final image, on the right the contents of the of the normals G buffer.

- **Texture 3:** Quarantamaula employs Mikkelsen's tangent space generation [3] in its model loader, so all four components are needed to store the tangent.
- **Texture 4:** The RGB components store the albedo or base color, while the fourth stores the amount and direction of anisotropy.
- **Texture 5:** In the red channel we store the metallic parameter, in the green channel the strength of the clearcoat layer, and in the blue channel the roughness of said layer.

In this engine the geometry stage is composed of two passes. The first one we already described, but the second is employed to draw what I dubbed the *shader*

mask. Since Quarantamaula supports different shaders in its lighting stage, there needs to be a way to differentiate which pixels will be drawn with which shader. That is what the shader mask is for.

Apart from the general GameObject vector, there's a list for each type of shader that will be later used during the lighting stage. These lists are filled with the GameObjects that will be drawn with that shader. During this second pass, we assign a color channel to each shader and loop through each vector, drawing the geometry of its GameObjects with the assigned color channel into the shader mask texture (Figure 4). Later, that texture will be sampled to assign which fragments of the screen correspond to which shader.

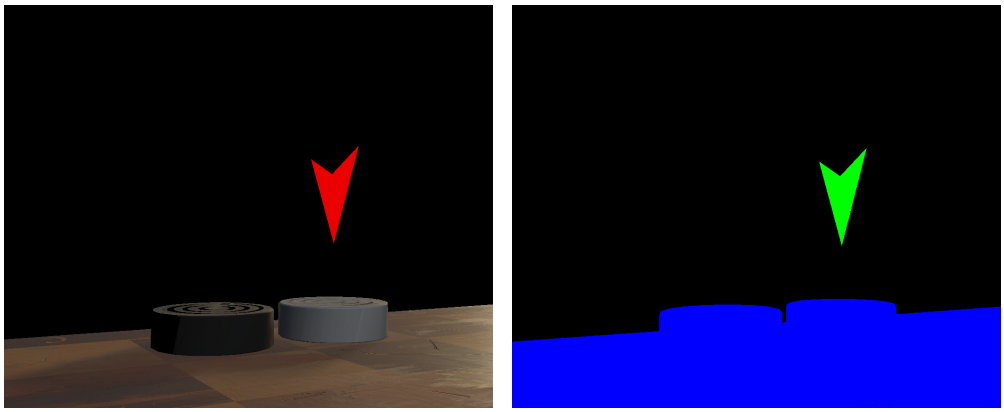

Figure 4: On the left the final image, on the right the contents of shader mask texture. The chess table and checkers are rendered with one shader, the arrow is rendered with another one.

As we can see in Listing 1, if we had more than 4 different types of shaders to use during the lighting pass, we would just attach another texture once we ran out of channels.

Listing 1: The shader mask pass

```
1  shaderID = ResourceManager::getShader("shaderMask")->getID();
2  glUseProgram(shaderID);
3
4  //Each shader mask is stored in a different channel of RGBA
       textures
5  for (unsigned int i = 0; i < NUM_SHADERS; i++)
6  {
7          //One-hot vector that determines the channel the shader
               mask will be stored in
8          glm::vec4 channel;
9
10         switch (i % 4)
```

```
11          {
12                  case 0:
13                          channel = glm::vec4(1.0, 0.0, 0.0, 0.0);
14
15                          //If we've filled out the texture's
                                channels, we bind the next framebuffer
                                 with a new texture attached to it
16                          glBindFramebuffer(GL_FRAMEBUFFER,
                                shaderMaskBuffers[static_cast<int>(std
                                ::floor(i / 4.0))]);
17                          glClear(GL_COLOR_BUFFER_BIT |
                                GL_DEPTH_BUFFER_BIT);
18
19                          break;
20
21                  case 1:
22                          channel = glm::vec4(0.0, 1.0, 0.0, 0.0);
23                          break;
24
25                  case 2:
26                          channel = glm::vec4(0.0, 0.0, 1.0, 0.0);
27                          break;
28
29                  case 3:
30                          channel = glm::vec4(0.0, 0.0, 0.0, 1.0);
31                          break;
32          }
33
34          //Send the channel to the shader
35          glUniform4fv(glGetUniformLocation(shaderID, "channel"), 1,
                 glm::value_ptr(channel));
36
37          for (auto& obj : shaderGameObjects[i])
38          {
39                  obj->drawGeometry(shaderID);
40          }
41  }
```

2.2 Lighting Pass

For this stage, Quarantamaula loops through the shaders, rendering a screen quad for each using that shader. We bind the G buffer textures that shader needs, along with the shader mask and any other needed textures (shadow or environment maps).

Then in the fragment code, before doing any calculation, the shader will sample the shader mask and look at its assigned color channel. If the value in the shader mask for that fragment in the assigned color channel is greater than 0, then that fragment corresponds to this shader. And so, the lighting calculation can begin. Otherwise,

it's left empty to be filled by other shader that comes afterwards, or the clear color or skybox if no shader occupies that fragment.

When each type of shader renders its screen quad, the quads are blended together and the final image is presented on screen.

Though, to be precise, there's an optional pass beforehand for gizmos. These are the x, y, z axes, and indicators for the direction of the directional light and position for the point lights. These are drawn with depth testing disabled so they're always rendered on top of everything else.

3 PBR

Among the shader types Quarantamaula supports for the lighting stage, the most noteworthy one is the PBR shader. PBR stands for Physically Based Rendering, that is, employing shading equations that aim to mimic the way materials in real life reflect and refract light.

The PBR shading model in this engine is mostly inspired by the one employed by Filament, an Android PBR Engine [2]. The BRDF f_r is composed of three components or layers blended together as shown in Equation 1.

$$f_r = (f_d \cdot \ k_d + f_s) \cdot \ (1 - F_c) + f_c \tag{1}$$

Where f_d, f_s and f_c are each of the lighting components which we'll delve into later, kd is the diffuse lighting coefficient and F_c the clearcoat layer's fresnel. The outgoing radiance L_o, before accounting for ambient light, is computed as seen in Equation 2.

$$L_o = L_i \cdot \ f_r \cdot \ (\vec{N} \cdot \ \vec{L}) \tag{2}$$

The computation of incoming light L_i is done through the simple analytical light transport models of directional and point lights, along with shadow maps.

3.1 Diffuse Component

The diffuse component f_d is a simple Lambertian BRDF (Equation 3).

$$f_d = \frac{baseColor}{\pi} \tag{3}$$

What's more interesting is the derivation of its coefficient k_d. When light interacts with a surface, a proportion of light gets reflected off the surface, and the rest refracts and enters the surface. The diffuse component represents local subsurface scattering. That is, light refracted into the surface, that bounces around, and comes out right next to where it entered. Since the fresnel term represents the proportion of reflected light, $1 - F$ represents the proportion of refracted light. So $k_d = 1 - F$.

But we have a metallic parameter for a reason. Metals behave differently. With metals, all light gets reflected, none refracted. We also need to take that into account, so the complete k_d term can be seen in Equation 4.

$$k_d = (1 - F)(1 - metallic) \tag{4}$$

3.2 Specular Component

The specular component represents the reflected light. In PBR specular BRDFs f_s are based off of microfacet theory. This theory aims to model the microscopic irregularities that the surfaces of materials present. It postulates that all surfaces are composed of minuscule faces or *facets* arranged in tiny mountains and valleys (Figure 5).

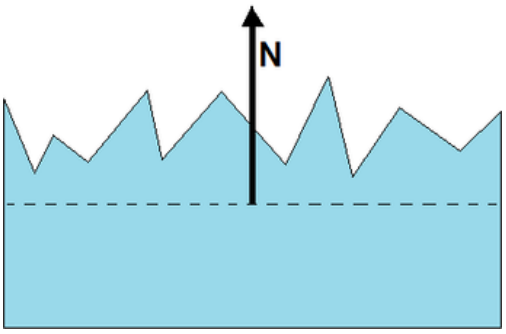

Figure 5: The microfacets arranged in mountains and valleys.

And these facets are considerd to be perfect mirrors, meaning, they reflect all light. These specular BRDFs based on microfacet theory follow a standard model (Equation 5).

$$f_s = \frac{D \cdot F \cdot G}{4 \cdot (\vec{N} \cdot \vec{L})(\vec{N} \cdot \vec{V})} \tag{5}$$

Let's break down what each term means:

- **D** is the statistical distribution of the microfacets. Since it would be unfeasible to model these irregularities with actual geometry, we model them through statistics. Specifically, the D term models the orientation of the microfacets by how close their normal vector is to $\vec{N}$, the surface's overall normal.
- **F** is the fresnel. It essentially acts the same way as k_d, except for the specular component. It's used to scale the specular contribution because it represents how much light gets reflected.
- **G** models the entrapment of light due to the geometry of the microfacets. Meaning, light that is headed for a microfacted in a valley but is blocked by the mountain next to it, and light that bounces off a microfacet towards the camera but is blocked by another mountain before reaching it.

Now let's get into the specifics of how this is implemented in this engine. Both D and G need to follow a specific statistical distribution. In Quarantamaula we use the most popular one, GGX. We also employ the universally used approximation of the fresnel equations, the Schlick fresnel. Lastly, we don't use the standard equation, instead opting for a deviation which incorporates the denominator into the G term for performance reasons. This is the visualization V term and it's the same as $\frac{G}{4\cdot (\vec{N}\cdot \vec{L})(\vec{N}\cdot \vec{V})}$. The result is shown on Equation 6.

$$f_s = D_{GGX} \cdot F_{Schlick} \cdot V_{GGX} \tag{6}$$

3.3 Clearcoat Layer

The clearcoat component f_c is meant to model a layer of reflective paint added on top of the base material. Think of the reflective finishes of automotive paint. As such, this layer only consists of a specular component (Equation 7).

$$f_c = D_{GGX} \cdot F_c \cdot V_{Kelemen} \tag{7}$$

Since this layer is less flexible than the base layer, always being isotropic and dielectric, with low roughness values, we can choose cheaper terms without notably sacrificing visual quality. So we employ a less computationally expensive V term with no physical basis.

F_c is equal to $F_{Schlick} \cdot clearcoat$, and it's used to control the strength of this component. It too balances the strength of this layer and the base layer, as seen in Equation 1, in order to maintain energy conservation. Because the light that gets refracted off this layer, $(1 - F_c)$, will enter the base layer.

3.4 Parametrization

Let's go over how we can control the look of this shader. The following parameters are utilized to define the properties of the material we're trying to model.

- **baseColor** is in physics terms the albedo. But it's pretty self explanatory, it's the color of the base layer.
- **roughness** defines how irregular the base layer's material's surface is. The higher the roughness, the less reflective a material appears because its microfacets are facing directions all over the place, scattering the diffuse reflection and blurring it. Lower roughness means more microfacets are aligned with $\vec{N}$, so the reflection is more concentrated around a single direction and sharper. To the D and G terms we don't pass the raw roughness as an argument, instead we remap it first. We pass $\alpha = roughness^2 + 0.001$.
- **specular** controls the base reflectiveness of the base layer's material. The Schlick fresnel approximation interpolates between a base reflectiveness F_0 and a peak reflectiveness F_{90}, depending on the angle of incidence. $F_0 = 0.16 \cdot specular^2$, while F_{90} is fixed to 1 (all light is reflected).

- **metallic** models whether the material is a metal or not. Metals also behave differently when it comes to the light they reflect. Unlike most materials, metals tint the specular reflection. So, apart from controlling the strength of the diffuse component via k_d (Equation 4), this parameter also plays a part in the computation of F_0. Its complete derivation can be seen on Equation 8.

$$F_0 = (0.16 \cdot \ specular^2)(1 - metallic) + baseColor \cdot \ metallic. \tag{8}$$

- **anisotropic** WIP
- **clearcoat** controls the strength of the clearcoat layer by scaling F_c (Equation 7). The base fresnel value of the clearcoat layer's fresnel F_{0c} employs a constant value of 0.04, that is consistent with the material this type of paint is made of most of the time.
- **clearcoatRoughness** is the exact same as roughness, but for the clearcoat layer instead of the base layer. It's assumed that this value will almost always be very low, since the point of the clearcoat layer is to appear very reflective.

4 Shadows

Shadows are an essential part of computing the incoming light L_i needed for the rendering equation (Equation 2). In Quarantamaula, they are achieved through the ol' reliable, shadow maps. This technique consists in rendering the scene from the point of view of the light source (or light sources), storing the depth information in a texture and then querying that data when doing lighting computations to see whether the fragment we're rendering is occluded from the light source. As such, shadows rendered with shadow maps also require two passes which we'll expand on now.

4.1 Shadow Map Generation

The shadow map pass, as stated before, consists in rendering the scene from the point of view of the light source with only a depth buffer attached. And then saving said depth buffer, which contains how far away each fragment is from the camera, as a texture. That texture is the shadow map that gives name to the technique. Here, we'll distinguish between how this is done for directional lights and point lights,

- For **directional lights** we render the scene placing the eye a set distance away from the world origin in the trajectory of the directional light's direction. We render the scene with an orthographic projection matrix. The specified view and projection matrices together form the *light space matrix*.

  Then we begin drawing only the geometry of all the GameObjects in a specific vector which contains those objects we want to cast shadows. Lastly, the depth information is stored in a texture (Figure 6).

Figure 6: The directional shadow map. Each pixel stores how far away it is from the camera in grayscale. Lower (darker) values mean it's closer, higher (lighter) values mean it's farther.

- For **point lights** this is a bit more complicated, since we need to capture the scene's depth information from all directions. We use cubemaps to achieve this. We employ 6 view matrices that correspond to having the eye facing each of the cube's faces. For projection matrix we employ a perspective projection with a 90 degree FoV, in order to capture completely the cubemap face, and only the face we're looking at. From these we get 6 light space matrices.

  The shader program used to render the point light shadow maps includes a geometry shader. This is to avoid having to render the scene 6 times, once writing to each face of the cubemap. Instead we do it all in one go, using the variable `gl_Layer`, which tells the API which cubemap face to render to. To the geometry shader we pass the 6 light space matrices and for each primitive generate 6 triangles, each transformed to the light space of it's corresponding face (Listing 2).

Listing 2: The point shadow geometry shader

```
1  #version 460 core
2
3  // Input is a triangle
4  layout (triangles) in;
5  // Output is 6 triangles (18 vertices), one for each face of
       the cubemap
6  layout (triangle_strip, max_vertices=18) out;
7
8  // Matrices used to transform vertices into the light space
       of each of the cubemap faces
9  uniform mat4 ptLightSpaceMats[6];
10
11 // Fragment's position in world space
12 out vec4 fragPos;
13
```

```
14 void main()
15 {
16     for(int face = 0; face < 6; face++)
17     {
18         // Set the cubemap face we're gonna render to
19         gl_Layer = face;
20
21         // For each of the vertices in a triangle pass along
               their world space position and transform them into
                light space
22         for(int i = 0; i < 3; ++i)
23         {
24             fragPos = gl_in[i].gl_Position;
25             gl_Position = ptLightSpaceMats[face] * fragPos;
26             EmitVertex();
27         }
28         EndPrimitive();
29     }
30 }
```

The GameObjects are drawn in the same manner as for directional lights and the cubemap with the depth information is stored.

4.2 Computing L_i

Now that we've got the shadow map, we can use it during the lighting pass to obtain the incoming light L_i. It's complete derivation is shown in Equation 9.

$$L_i = lightColor \cdot \ shadow \cdot \ a^* \tag{9}$$

Where a is the light attenuation, and it's marked with an asterisk because it's only present on point lights.

The more interesting term, however, is $shadow$, which represents occlusion from the light source. If it equals 1, it means that this fragment is in line of sight of the light source. 0, on the other hand, means that no light gets to this fragment because there's geometry in the way.

In order to obtain this value we must sample the shadow map for this fragment. If the depth value of our fragment is greater than the one we sample, it means this fragment is in shadow and $shadow = 0$. Because it means that, from the point of view of the light source, there's something in front on our fragment.

But, in order to make this comparison make sense we first need to transform our fragment to the coordinate space of the shadow map. Which means, transforming this fragment from as being seen by the camera to as being seen by the light source. In order to do this we simply multiply it's world-space position by the light space matrix mentioned earlier.

That is in the case of the directional light. For point lights it's much simpler. We get the vector from this fragment to the light source, use it to sample the cubemap, and compare the sampled value with the length of the vector.

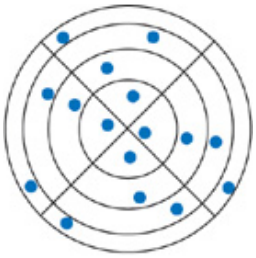

Figure 7: An example of what a shadow sampling pattern looks like.

4.3 Shadow Filtering

So far, with our binary approach to $shadow$ we would get hard shadows. But these are often not desirable, as they don't look realistic and, unless you've got a shadow map with a really high resolution, there will be visible texels in your shadows. In order to get nicer looking soft shadows we apply random sampling.

To get a softer transition between shadow and light, instead of sampling the shadow map at just one point, we sample multiple texels from the shadow map around the point that corresponds to the fragment we're currently rendering, and average the shadow value we obtain from them.

And to achieve even more realism and avoid visible patterns in the shadows, the points around the fragment where we sample are randomized for each fragment. Not completely randomized though, we use stratified sampling. We employ a set of concentric rings divided in 4 sections, with a sample in each section (Figure 7).

In order to get these patterns in the shader, we first generate a 3D texture of random samples, offline. Then when rendering, we map each fragment to a cell in the texture, and in that cell we find stored the array of random samples that constitutes a sampling pattern for that fragment.

One last detail I want to mention is that, to reduce the number of texture fetches we need to perform, we first test the samples in the outer ring. If they all come back with the same value (either in shadow or not in shadow), we just return that value and not check the inner rings.

This is an extensive algorithm, so I've omitted the finer implementation details for brevity's sake. If you'd like to know more, you can learn directly from the source [5]. The algorithm in Quarantamaula is mostly the same as the one presented there.

The one thing in which they differ is that this engine also has support for shadow-casting point lights, so the filtering needs to be applied there too. This is fairly simple though. To sample points around the one that corresponds to the fragment we offset the sampling vector in the xy axes by the values stored in the sampling pattern (Listing 3).

Listing 3: Adapting filtering to point light shadows

```
1 // Sample the shadow map at the first sample point
```

```
2  sampledDepth = texture(ptShadowMaps[index], direction + vec3(
       filterMapTexelContent.rg, 0) * vec3(texelSize, 1)).r;
3  // Transform it from the [0,1] to its original range
4  sampledDepth *= PL_farPlane;
5  // If the current depth value is greater than the sampled one,
       this sample point is in shadow
6  shadow += (currentDepth - SHADOW_BIAS) > sampledDepth ? 0.0 : 1.0;
7
8  //Repeat the process for the other sample point
9  sampledDepth = texture(ptShadowMaps[index], direction + vec3(
       filterMapTexelContent.ba, 0) * vec3(texelSize, 1)).r;
10 sampledDepth *= PL_farPlane;
11 shadow += (currentDepth - SHADOW_BIAS) > sampledDepth ? 0.0 : 1.0;
```

5 IBL

Needs a little more work before I can document it properly. Here, in Figure 8, you can see what it can do.

Figure 8: Image based lighting teaser.

6 Model Loader

An engine is nothing without 3D models to render. That's why incorporating a good model loader is key. Quarantamaula can load wavefront (.obj) files, with a little add-on.

6.1 Vertex and Index Data

Models are loaded by supplying the filepath of the .obj file we want to load to the Model class constructor. The loader then loops through each mesh in the model, going through the faces specified in the file, which must be triangular. When it's finished, before assembling the mesh with the vertex and index data obtained from the file, we generate the tangents to complete the vertex data (Listing 4) using the ubiquitous MikkTSpace [3].

Listing 4: The structure of the vertex data passed along to the geometry pass shader.

```
struct Vertex
{
        glm::vec3 position;
        glm::vec3 normal;
        glm::vec2 texCoords;
        // w is MikkTSpace handedness
        glm::vec4 tangent;
};
```

Assembling the mesh entails passing the vertex and index data to the Mesh class constructor, which uploads this data to the GPU, creating the appropriate VAO, VBO and EBO. It follows, then, that the Model class contains a vector of Mesh objects.

6.2 Textures

When encountering a material reference in the .obj file, the loader searches for the .mtl file and begins parsing it. From .mtl files the only feature we care about and load is the textures they reference. These are loaded into a vector of textures in the Model class, and then each Mesh is assigned textures from this vector. That way, if two meshes employ the same texture, it's not loaded twice.

Per object material parameters in the .mtl file (kd, ks, etc.) are ignored, because they are already provided when constructing the object in the engine's code. These per object parameters act as a fallback if no texture data is loaded for that parameter. For example, if there's no specular map defined in the .mtl file, we'll employ the value specified for the $specular$ parameter in the object's constructor. However, if the material defines a specular texture, said texture will be loaded and written into the G buffer as opposed to the per object value, which will be ignored.

Similarly, if no normal map is specified in the material file, we employ the normals obtained from the vertex data.

The little plus mentioned earlier is that apart from supporting the diffuse, normal and specular maps, which are wavefront standards, we also support textures for the

$roughness$, $metallic$ and $clearcoat$ parameters. These must be specified in the .mtl file via the keywords `map_roughness`, `map_metallic` and `map_clearcoat`, respectively, followed by the path to the corresponding texture.

References

[1] Joey De Vries. *Learn opengl: Learn modern opengl graphics programming in a step-by-step fashion*. Kendall Welling, 2020.

[2] Romain Guy and Mathias Agopian. URL: https://google.github.io/filament/Filament.md.html (cit. on p. 6).

[3] Morten Mikkelsen. URL: http://www.mikktspace.com/ (cit. on pp. 3, 14).

[4] Tomas Möller. *Real-time rendering, fourth edition Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michal Iwanicki, Sébastien Hillaire*. CRC Press, 2019.

[5] Yury Uralsky. *Chapter 17. efficient soft-edged shadows using pixel shader branching*. URL: https://developer.nvidia.com/gpugems/gpugems2/part-ii-shading-lighting-and-shadows/chapter-17-efficient-soft-edged-shadows-using (cit. on p. 12).