After implementing the simple and empirical Phong illumination model in the first lab, this time we will follow a physical approach considering the behavior and properties of each light and material. The two main factors within the physically based rendering model (PBR) are implementing it for direct lighting and using *Image Based Lightning* (IBL) for indirect lighting.

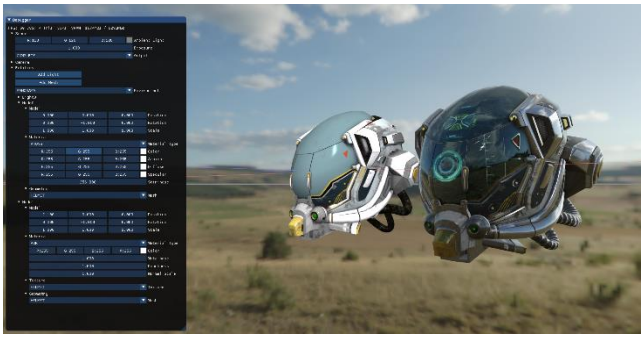

Figure 1: Difference between PBR (right) and Phong (left) on helmet mesh with the default values

The first thing we did was creating a subclass *PBRMaterial* that inherits from *StandardMaterial* and overwrites all its methods (which we will explain later), but before let's dive into the attributes of the material, being the first two *metallic_texture* and *roughness_texture* to store que corresponding metalness and roughness textures of the material, having added four extra maps: *normal_texture* to store the normal map that gives details to the surface of the corresponding mesh defining the illumination of every pixel, *emissive_texture* to store the emission map that adds the light emitted from the material itself, *ao_texture* to store the ambient occlusion map that simulates the mesh self-occlusions with ambient light and, finally, *opacity_texture* to store the opacity map that indicates how transparent are the different parts of the mesh. Apart from the actual textures, we have the boolean *metallic_roughness* to indicate if the metalness and roughness are in the same texture and, finally, we have *roughness* and *metalness* that are factors to modify the corresponding texture values.

Before diving into the details, we must focus on some changes done to some classes that already appeared in Lab 1. First of all, for IBL we cannot still use the skyboxes from Lab 1 because they are defined in the range $(0, 1)$ and we need HDR textures with range $(0, \infty)$ to correctly represent the light values as in the other case are clamped, so when we create the skybox node (or it is changed) we load the corresponding HDRe and convert it to the corresponding cubemap using *cubemapFromHDRE()*, indicating the level 0 because in this case we don't want it blurred. That being said, we need blurrier versions of an skybox for IBL because of the effect of the roughness of the material, so 5 more attributes are created storing the cubemap corresponding to each level of blurring. Secondly, we cannot still use the light diffuse and specular colors from *Phong* equation, so we define the attributes *color* and *intensity* for PBR.

Going back to the *PBRMaterial* class, in the constructor we set the default values to correspond with the *helmet* mesh because it's the default one loaded as *ScreenNode*, where in *setUniforms()* we first set the common uniforms from *StandardMaterial* and, after that, all the PBR attributes are uploaded to the shader, where each texture is in a different slot to avoid that a texture is overwritten to another and we pass the different *skybox* textures too.

Now, the *render* method has changed from the one used in *phong*, being the main motive the HDR rendering and the need of a tone mapper, which is for the corresponding HDR render to map its colors to SDR and be displayed in a common monitor, trying to maximize the perceptual similarity to the original one. That being said, tone mapping has to take into account all the illumination in an object but before the gamma correction (more on this later), which using the previous multi-pass renderer could not be applied when using more than one light, so the render implemented is a single-pass one, which stores all the light information in arrays (position, color and intensity) and uploads them to the shader using a single draw call, having a maximum of 100 lights.

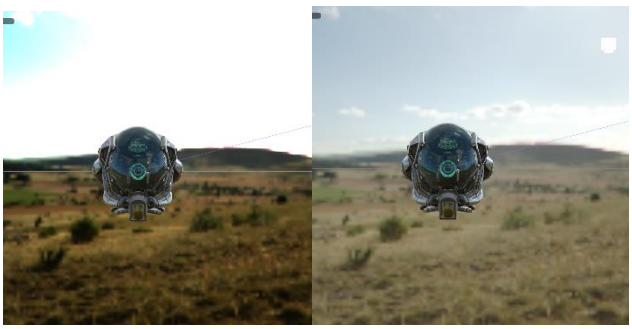

Figure 2: Difference between applying tone mapping and gamma correction (right) and not applying them

Finally, to have support for transparent materials blending is activated using a blending function of GL_SRC_ALPHA and GL_ONE_MINUS_SRC_ALPHA where on the source is the α and in the destination is $1 - \alpha$, where we need to only render the front face to avoid

artifacts while rendering transparent surfaces setting GL_CULL_FACE to GL_FRONT and indicating GL_CW to render the different triangles in order.

Figure 3: Transparency in lantern mesh

The *PBRMaterial* can be changed in ImGUI where there is the color (with the alpha channel for transparency), the metalness and the roughness. Finally, and abstract method was created in *Material* that was overwritten in *StandardMaterial* and *PBRMaterial* that selects the correct textures depending on the name of the mesh to render and considering its number to know if roughness and metalness are in the same texture.

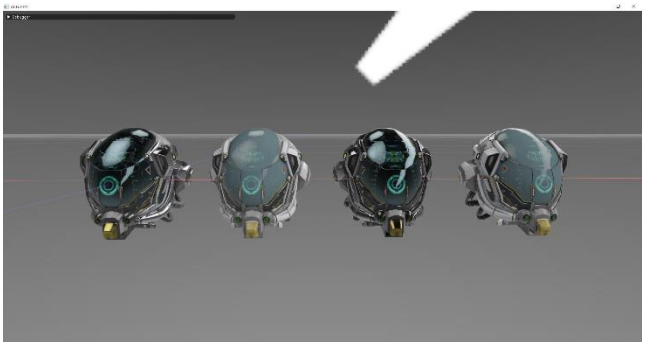

Figure 4: Full PBR render (first), no metalness (second), no roughness (third), no metalness and roughness (last)

Diving now in the PBR shader, we start by creating and filling a struct that represents the PBR material in that fragment containing the different values related to it, where this values are extracted in the function *getMaterialProperties()* where the texture values are read using *texture2D()* because they come from 2D textures and using the UVs (*v_uv*) interpolated from the vertex shader, where in the case of roughness, metalness, ambient occlusion and opacity we only get one concrete channel as they are float values and not colors. Another thing we must consider is that there are certain values that are given in logarithmic space since this is the space in which most monitors work. So all RGB values that are entered by a human user looking at a monitor like albedo and emission textures and light color and color factor as they are selected from the colors indicated in ImGUI displayed through a non-lineal monitor will be stored in logarithmic space and having to linearize them before use them in any computation using the function *gamma_to_linear()* and, at the end of the shader when all the calculations have been done, we transform the final color to logarithmic space to be displayed on the screen. The normal have to be perturbed with the function *perturbNormal()* to transform the normal map values from tangent space to object space, where we used $-\boldsymbol{V}$ because looking at the result in the HDR4EU demo and Unity, we saw that the normals were inverted.

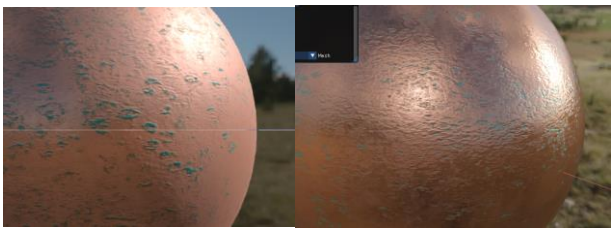

Figure 5: Comparison between normalmaps in HDR4EU demo (left) and in the application using V instead of -V (right)

Once this process is finished we can move on to *getPixelColor()* where we first compute the indirect light using IBL, where in the specular part we get the reflection color using *getReflectionColor()* with the reflected vector $\boldsymbol{R}$ and the roughness, where vector $\boldsymbol{R}$ indicates where in the skybox our "view ray" bounces off from the mesh at that point. For this we need to reflect the vector $-\boldsymbol{V}$ (negative because we need it to go in the direction of our eye to the point of the mesh in the world space) based on the vector $\boldsymbol{N}$. The result will be a vector that goes in the direction from the point of the mesh to the point of the skybox that we have to reflect, ready to be read using *textureCube()*. Related to applying the specular BSDF we use the function *FresnelSchlickRoughness()* and the BSDF pre-computed in a LUT, where while the texture coordinates are not defined in 0 nor 1, we used GL_CLAMP_TO_EDGE when the LUT texture is created so we do not need to clamp the UVs used ($\boldsymbol{N} \cdot \boldsymbol{V}$ and roughness). For the diffuse part, we use $\boldsymbol{N}$ in the reflection color and the maximum roughness, considering the conservation of energy of $(1 - K_s)$. Finally, specular and diffuse components are added and multiplied by the ambient occlusion as it only affects the indirect light.

Continuing on the direct lightning, we defined a *for loop* with the maximum of lights that will break when

the last light has been applied, doing this because the GPU is not the best handling *for loops* and this is more efficient. That being said, we first define the direct light as (0,0,0) and we will add the result of the different lights, defining the vectors and dot products that depend on the light that is being calculated. Where we will have to compute the BSDF for each of them, starting by calculating the diffuse part using the Lambertian model and later computing the specular part, where we will need to use the Cook-Torrance model, in which we will compute the F (Fresnel), G (geometric attenuation due to microfacets), and D (normal distribution function) terms and the dot products of the vectors that affect each one. For the latter we will need the vector $\boldsymbol{H}$, a new vector introduced in this lab, being the half between the vector $\boldsymbol{V}$ and the vector $\boldsymbol{L}$. Once these calculations are finished, we will divide by the normalizing factor. The computed direct light will be the sum of the diffuse and specular parts multiplied by the light color in lineal space, the intensity and $\boldsymbol{N} \cdot \boldsymbol{L}$ to define the surface where the light is applied.

Figure 6: Helmet mesh with color (134, 81, 101, 194) and light in position (10.0, 0.5, 21.0) with color (140, 163, 51) and intensity 35.0

Finally, the final color will be the sum of all the calculated direct lights and the indirect light plus the material emission (as it is like another light), where this color will be tone mapped with the Uncharted tone mapper and, finally, converted to gamma space again to be displayed. The final color indicated in gl_FragColor has a fourth component that is the opacity indicated in the corresponding texture that will be multiplied by the alpha channel of the color factor. Finally, we must comment that the skybox shader has been changed adding the tone mapping and the lineal to gamma passes after getting the final color (the HDRe cubemaps are already in lineal space) to correctly display them, where the same has been applied to the reflective shader.

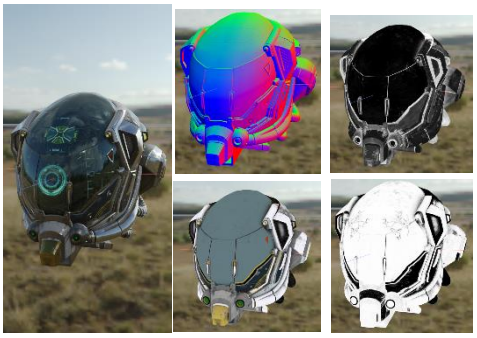

Figure 7: Full PBR render (left), albedo texture (center below), normal map (center above), roughness map (right above), metalness map (right below)

As additional feature, we added the *distance_to_cam* attibute to the *SceneNode* class that indicates the distance to the camera of the mesh and implemented a sorting function for the vector that contains the scene nodes to order the different nodes by distance so that if an opaque object is before a transparent one, you can see it through the transparent one. Moreover, it's important to consider the cases where a certain extra map is not given, where in this situations we have to substitute them, where for the emissive map we pass a black texture so nothing is added, for the ambient occlusion we pass a white texture so after multiplying the result is not altered and, finally, if there´s no opacity map, we use a white texture so the object is completely opaque. Finally, all textures and the LUT (stored as an *Application* attribute) are loaded at the beginning to have a faster change between meshes.

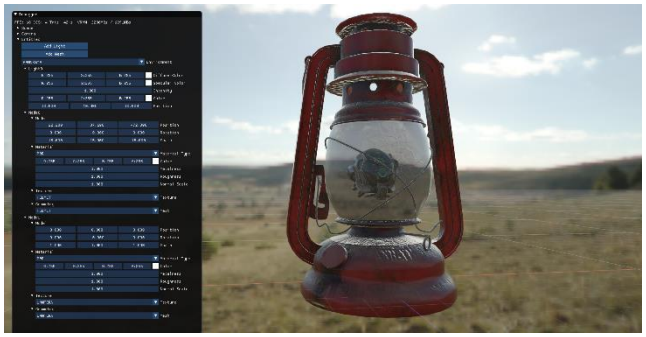

Figure 8: Helmet mesh transparented through lantern mesh

Now it's time to try it yourself in the application!