This being the reintroduction to concepts that were learned throughout Computer Graphics' course, this first lab has been centered in working on the concepts of nodes, materials, lights and shaders, performing the rendering of a simple scene taking into account diverse concepts such as lighting using Phong's equation, texturizing materials, or even working on applying a skybox. The chosen structure has been to have all meshes, lights and skybox as nodes of the scene where each group is stored separately, that is, meshes and lights in its respective *std::vector* and the skybox as a variable (there can't be many skyboxes). Since we have applied a multi-pass render, it is considered more agile to store everything separately at the beginning than to split an *std::vector* with all nodes of the scene including meshes, lights and skybox every frame rendered.

Once this has been explained, we define the default scene: the ambient light and a skybox, as well as a light and a mesh, where all of them can be modified once inside the application. We have placed the skybox renderer inside the general rendering of the application, just before rendering all the other nodes. We have done it without performing the depth test so that it is not painted in front of any object.

That being said, the next thing we performed was the simple task of applying a texture to an specific mesh, without complicating it with lighting or reflections. To do so, we have created a subclass *TextureMaterial* that inherits from *StandardMaterial* without overwriting any of its methods, changing only the fragment shader used going from *flat.fs* to *texture.fs* created by us, because no new uniforms were needed and no changes had to be made to the render.

Regarding the *texture.fs* shader, it is the one in charge of applying the texture corresponding to the mesh, where we send as uniforms the color of the material (in case it has been modified) as well as the texture itself and we use the values of the texture coordinates *v_uv* provided by the vertex shader (*basic.vs*) to read the specific texel with the *texture()* function to define the final color as the color of the texture multiplied by the color chosen by the user, where white corresponds to the texture without changes. These are the results:

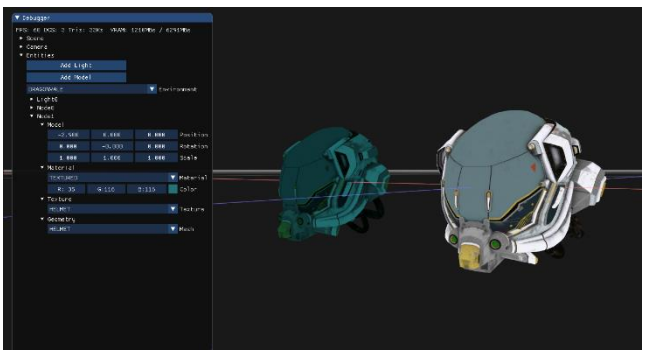

Figure 1: Two helmet meshes where the one on the left has been texturized and applying a color (35, 116, 166), while the one on the right has been texturized with a color (255, 255, 255), therefore without any change.

Next, we focused our efforts on defining an skybox for the scene, where we created a subclass of *SceneNode* called *Skybox* that overrides the *render()* and *renderInMenu()* methods. To apply a skybox, we decided to use cubemaps on a mesh that consists of a cube, using directories containing the images corresponding to each face of the cube and building the cubemap texture using the *cubemapFromImages()* function. On the other hand, to avoid that the camera moves out of the cube, the render adds only a code line from the *SceneNode* render placing the cube in the position of the camera using its model matrix and applying the *Matrix44* function *setTranslation()* to translate it to the camera eye. Finally, it is important to emphasize that the skybox can be changed using *ImGUI*, having available all the ones that are in the environments directory.

In order to paint the skybox, we cannot use the shader that we previously used because now we are working with cubemaps, which have a different reading. Therefore, we have defined a second shader, not much more complex but more interesting. In this case to read the cubemap we use *textureCube()*, where we cannot use the UVs coming from the vertex shader (*basic.vs* again) so, to know which position of the texture has to be read, we use a vector. Therefore, based on the unit vector V which is defined starting at the position in world coordinates that we want to paint and pointing towards the eye (the camera), we will look for its inverse vector, i.e. the one that goes from the eye to the position to paint, defining therefore a unit vector $-V$ to obtain the color at the point that we want to paint. Here we can se the results:

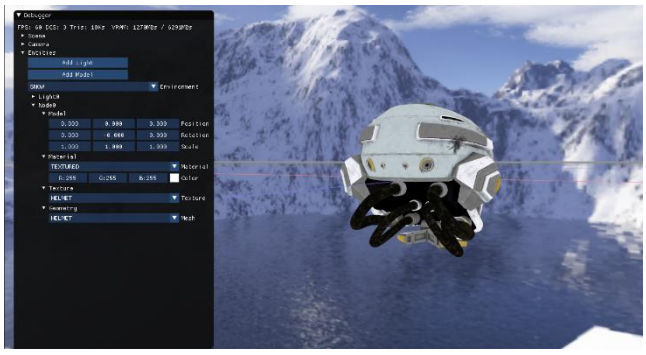

Figure 2: Snow skybox applied with a textured helmet mesh

Once we had the skybox for the scene, the next step for us was to render the mesh as a mirror reflecting the skybox of the scene. In order to do this, we have added another material creating the subclass *ReflectiveMaterial* that inherits from *StandardMaterial* and where it is not necessary to overwrite any method because it can perfectly make use of all the ones defined in the parent class. The shader used to do this is *reflective.fs*, where the structure is similar to the *skybox.fs* one, but we have another vector to take into account: the reflective vector R. The reasoning behind this is that whereas previously we took into account the vector from the eye to the specific skybox point, now we need the material to

reflect the skybox based on the view angle we have, therefore, we need $-V$ reflected on the surface of the mesh relative to the normal N at that point in order to obtain the unit vector R, that will indicate the unit vector needed to index the cubemap and obtain the corresponding color. The reflections are like the following:

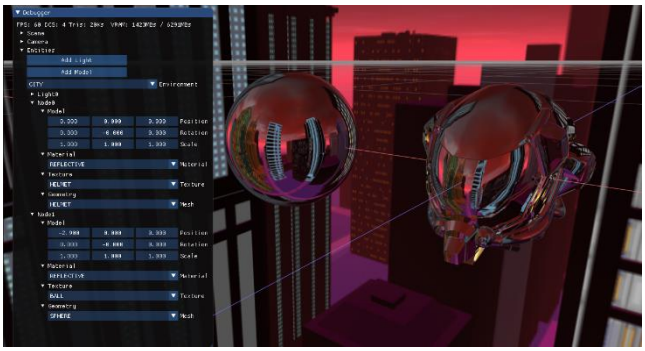

Figure 3: Sphere and helmet type meshes with a reflective material applied reflecting the City skybox.

We continue now with the first appearance of lighting following Phong's equation where we have created a subclass *PhongMaterial* which inherits from *StandardMaterial*, adding four attributes that describe how the material behaves regarding the following lighting factors: *ambient* in relation to the ambient light, *diffuse* with the diffuse component of the light, *specular* for the corresponding specular component and, finally, *shininess* to grade the power of how much the light affects the specular factor. In addition, the *render()*, *setUniforms()* and *renderInMenu()* methods have been overwritten. Furthermore, we have created a subclass *Light* that inherits from *SceneNode* with attributes *diffuse* to indicate the diffuse component of the light and *specular* for the specular one, having an ID named *lastLightId* to keep a count of the scene lights. The only method overwritten is *renderInMenu()* to change the light parameters.

Regarding the *PhongMaterial* overwrites it is very important to note the render where, as mentioned at the beginning, we have used multi-pass rendering, in which we render each mesh once for each light applying an additive blending *GL_ONE* so that it always adds up, not considering the alpha for the moment. In addition, in order for the ambient light to be taken into account only for the first pass, it is indicated as $(0, 0, 0)$ from the second light onwards. Related to the uniforms, the overwritten function uses the one from *StandardMaterial* because they share some uniforms and adds the ones related to upload the material properties already mentioned.

To apply the equation, we have the shader *phong.fs*, describing the equation at fragment level and not vertex to obtain a better final quality. The procedure is quite simple and follows the equation itself, which is already quite clear, where the following structure starts from initially reading the texture (which can change color as indicated by the user as earlier mentioned) where the resulting color will be multiplied by the different properties of the materials to have calculated the corresponding Ka, Kd and Ks. The rest are the consequent multiplications and scalar products between the different components of the equation, where are defined on one hand the vector L, starting from the world position that is being rendered towards the world position of the light in question and on the other hand the already mentioned vector V, with the R vector in this case being the one that reflects the incident vector of the light $(-L)$ and not the $-V$, establishing the shininess of the material as the power to which is elevated the scalar product between the vectors R and V to the calculation of the specular component. The applied Phong equation looks like this:

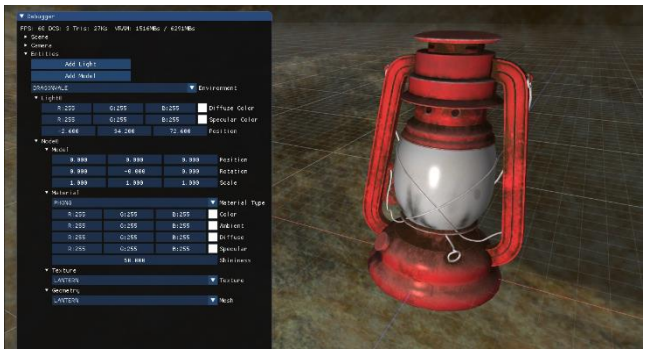

Figure 4: Lantern mesh using a material applying Phong's equation where the light has been translated to (-2.6, 94.2, 72.6) and the shininess has been changed to 50

Finally, we will dedicate a few lines to comment on the virtues of *ImGUI* and the possibilities that we implemented as well as its representations. Focusing first on the possibilities at scene level, where we have added the changes to the *main.cpp* file and we enabled changing the ambient light represented as a color to facilitate its visualization, included buttons to add lights and meshes increasing the complexity of the scene according to the user's preference, as well as the possibility of changing the skybox already mentioned. Here we can look to some results:

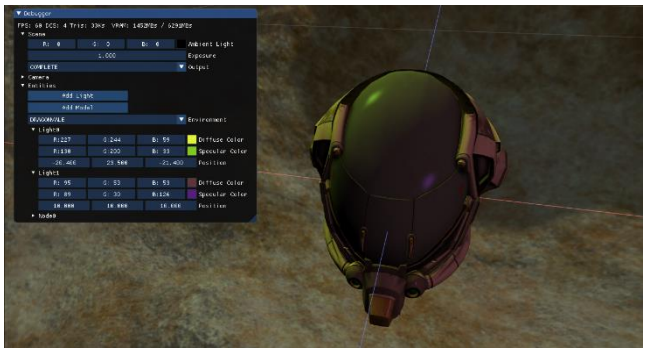

Figure 5: Helmet mesh with two lights and ambient light at (0, 0, 0). The first light with diffuse component (227, 244, 59) and specular component at (130, 200, 33) and position (-20.4, 23.5- -21.4). The second light with diffuse component (95, 53, 53) and specular component at (89, 30, 126) and position (10, 10, 10).

If we move down to node level, we can observe the possibilities regarding the changes seen in the images previously shown, being able to change textures, materials

and even the mesh itself. In all these cases the options are defined as combos because they are closed options. However, there is still material level which also contains a good amount of customization options, where considering that it is what we have added we will focus on the options regarding *PhongMaterial*, since the other types do not allow changes as the material properties do not intervene. In this case we will find that we can customize the properties of the material described as colors (except the shininess) because similar to the idea of the properties of the light, we believe that it is a much more comfortable representation for the user to see them represented like that and helps to have the values in the range (0,1) that has to be taken into account in the shader. These are the results:

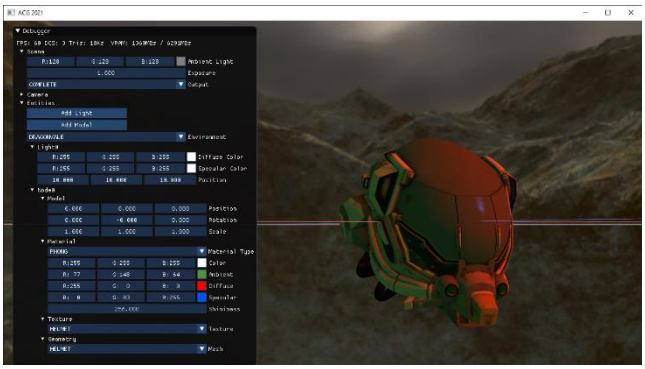

Figure 6: Helmet mesh with diffuse component (255, 0, 0), specular component at (0, 83, 255) and ambient component at (77, 148, 64).

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