Shadow Mapping Practical Work

IGR202 - Computer Graphics & Virtual Reality

Lais Isabelle ALVES DOS SANTOS

December 2022

1 Introduction

This report aims to briefly explain the steps of the Shadow Mapping of a scene composed of a back-wall, a floor and a rhino. To compile and work as expected, the project files and directories were refactored from the provided ones. The following commands from the *src* directory should be done to properly set the compilation and running.

```
$ mkdir build
$ cmake -B build
$ cd build; make
$ cd ..; ./tpShadow
```

2 Normal Mapping

The first part of the project is to add a normal mapping to the wall behind the rhino. Instead of just adding a texture, add a normal texture before it allows the scene to become more realistic. In this case, the texture to be added seems to be made of wood, which is not regular and straight in real life. Thus, the normal mapping will make the wall appears to be real wood.

In the code, the first part that was edited was *main.cpp*, specifically in the *initScene* function. The normal mapping can be treated as a texture, like it was coded for the practical work of the last period. In the case, the normal texture id was obtained by the *loadTextureFromFileToGPU* function, responsible for preparing the texture. To correctly bind and activate the texture, these code steps should be inside curly brackets, otherwise the texture will not be considered bound for the scene.

The following edited function was the *render* function, present in the *scene* class. This function is responsible for sending data to the shaders. As the normal texture should be rendered only for the back-wall, a Boolean variable, "material.useNormalMap" was created to identify whether it is needed to be used or not. The normal map is sent to the shaders by "material.normalMap" with its correspondent texture value on the GPU.

In the *fragmentShader*, the Boolean variable is checked and, if the considered render is the back-wall, the "albedo" variable is assigned to the normal texture. Otherwise, the "albedo" is the "material.albedo" sent from the *render* function. The figure 1 shows the result.

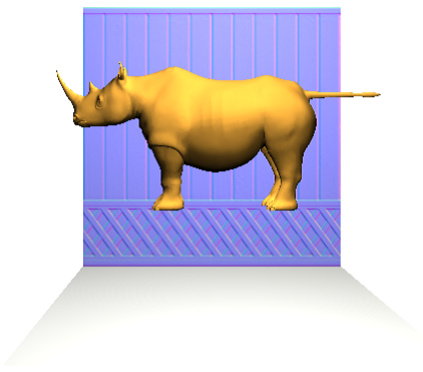

Figure 1: Normal Mapping

3 Texture Mapping

The editions made for the texture mapping were extremely the same as the ones made for the normal mapping. One particularity is that, instead of assigning the back-wall texture to the "albedo", it was created a new variable called "material.colorTex", that is passed to the shaders. Making use of the same 'if' condition of the normal mapping, the variable "text" in the *fragmentShader* is equal to the back-wall texture or equal to 1, depending on the actual render. The result can be seem in figure 2.

4 Shadow Mapping

For both shadow map generation and shadow rendering, [1] and [2] were key tutorials for obtaining the successful result for this part of the practical work.

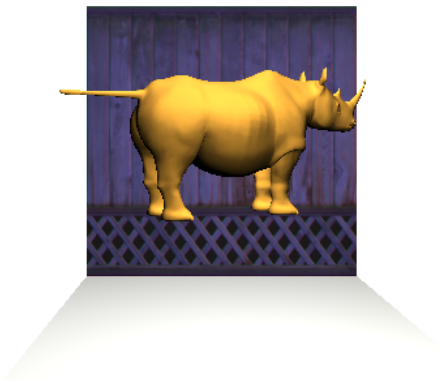

Figure 2: Texture Mapping

4.1 Shadow Map Generation

To generate the shadow map, it is required to, first, create and bind the depth map. This part was already done in the provided *main.cpp* file. The responsible class is *FboShadowMap*. The next step consist in creating the "depthMVP" matrix, a matrix mapped from the light point of view to a buffer. It means, this buffer (shadow map) contains the scene closest pixels to the light and they are reflected as non-white pixels, which can be seem in the figure 3 generated from 'ppm' files.

The first edited function was *setupCameraForShadowMapping*, present in the Light class. The camera is set for each light in the scene and so is created one depth matrix for each one of them. This matrix is composed of the multiplication of the projection matrix and the view matrix, because the shadows must be rendered close to the current scene. Otherwise, it could be mapped far away from what it is supposed to be seen.

The projection matrix was calculated with the support of '*glm::ortho*' function, using as parameters the ratio of the scene and the its center, but as it is set to zero at the Scene class, it was not put on the calculation. The z maximum and minimum values was taken from [1] and they worked fine.

To test the generation, the 'gl_Position' variable in the *vertexShader* was set to what the depth matrix was sending from each render multiplied by the 'shadowModel', a variable that is also sent to the fragments, but it is equal to one. The *fragmentShader* remained the same. Regarding the confirmation, the 'ppm' files played an essential hole in this part of the TP.

Figure 3: Generated shadows

4.2 Shadow Rendering

The shadow rendering was the most challenging part. Some different renders were obtained during the coding processing, like no shadow, shadows mapped in some weird places and repeated shadows along the scene. To render the shadows, in general, the lighting part of the *render* function in the class 'Scene' was edited, adding the components from the camera point of view. In this step, the shadows are sent as textures to the *fragmentShader* and each depth matrix is sent to *vertexShader*. The treatment applied to the depth matrix is close to that made in the shadow generation, but in this case, the normal matrices are used to compose the camera point of view. The shadow generation is now present in the *vertexShaderShadowMap*.

The shadow itself is calculated from the *shadowMapCalculation* function in the *fragmentShader*. The basic idea comes from the fact that the current fragments from the light rays are in the shadow only if they are greater than the closest fragments (the ones right in front of the objects present in the scene). As it is a shadow, to pass this parameter to the radiance, it should be the inverse of the result obtained from the function.

The problems mentioned before are due to the wrong variables used in the shadow calculation function. A simple debugging resolved them. As for the acne problem, caused when the surfaces of the objects are self-shadowed. To eliminate that, a bias is added so that the shadow samples from the shadow map no longer match with the surface. The acne result and correct result are shown in figure 4.

(a) Acne result (b) Final result

Figure 4: Shadow rendering

5 Conclusion

In conclusion, the practical work reported in this document allowed the practicing of the most important concepts involved in shadow mapping and the obtained result works as expected.

6 References

1. **Tutorial 16: Shadow mapping** http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/shadow-acne

2. **Shadow Mapping** https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping