UNIVERSITY OF LEEDS

Assignment 4 Report

Jingxuan Liu

Submitted in accordance with the requirements for the degree of MSc. High Performance Computer Graphics and Game Engineering

The University of Leeds

Faculty of Engineering

School of Computing

May 2023

Intellectual Property

The candidate confirms that the work submitted is his/her own and that appropriate credit has been given where reference has been made to the work of others.

This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

© 2023 The University of Leeds, Jingxuan Liu

Contents

Chapter 1

Task 1 - Shadow Mapping

1.1 Loading mesh and textures

The procedure of loading the mesh follows my implementation in CW2. The main codes for this process is in `vertex_data.cpp/hpp`, where the mesh information will be stored in a `TextureFragment` structure, possessing all the vertex attributes and material ids.

1.2 Shadow render pass

The logic behinds the shadow mapping is to create two render passes: the first pass will be used to render a depth buffer in the view of light, the second pass will used this depth buffer along with other relevant textures to apply BRDF algorithm. The creation of the shadow pass happens in the function `create_shadow_pass` in `main.cpp`. The shadow pass will take in the only one attachment, namely the shadow buffer, which has the format `VK_FORMAT_D32_SFLOAT` single channel. I had manually defined the dependencies, describing the pass will depend on external source and some external source will depend on this pass.

1.3 Shadow pipeline and pipeline layout

The shadow pipeline will take the only descriptor layout: the uniform scene uniform. The shadow pipeline will have 2 stage: vertex shader stage and fragment shader stage, where the first stage will transform the vertex position into the light space and the latter will be remained empty, just for recording the depth value. Therefore, in the vertex shader stage there will be

only one vertex attribute input, the position. I had manually defined the `depthCompareOp` to be `VK_COMPARE_OP_LESS_OR_EQUAL`. Moreover, in the `VkPipelineRasterizationStateCreateInfo`, I had declared some following codes:

```
 rasterInfo.depthBiasEnable = VK_TRUE;
rasterInfo.depthBiasConstantFactor = 0.005;
rasterInfo.depthBiasSlopeFactor = 0.01;
```

These codes will allow the renderer to 'push' the polygon a little bit further, in which will increase the depth value by a small amount and mitigate the self-shadowing.

1.4 Shadow depth imageview and descriptor

The default resolution of my shadow depth image is 1024x1024. After the successful creation of the image and imageview, they will be bind to the descriptors of individual mesh, so that they could be sampled by all the meshes in the rendering stage.

1.5 UBO buffer

Apart from the necessary parameters for calculating BRDF, a new matrix, recording the transformation matrix to light space will be passed to the shader. The matrix can be get by concatenating the light projection matrix and the light view matrix. By default, the light will have a FOV of 120 radians, positioned at (0.f, 5.f, 7.f), looking at (0.f, 0.f, -100.f) with an up vector (0.0f, 1.0f, 0.0f). These choices is the result of many attempts, aiming to get the best visual effects and efficiency. These settings will make sure the light looking at the status and with a wide visual field, so that the depth image will include more meshes and allowing more shadow calculation during rendering stage. Figure **??** image shows the depth image result from the first render pass.

The grey parts in the figure represents the closest distance the light hit on the mesh.

1.6 Rendering pipeline

Overall, the rendering pipeline was implemented based on the previous implementation in CW2, as well as adding features of the shadow mapping. In the vertex shader, the vertex position

Figure 1.1: depth image result

translated into light space will be calculated and passed to the fragment shader. In the fragment shader, the vertex position in light space will first be applied with the perspective division and then be normalised into range [0, 1]. Then, a bias will be added to the coordinates, the value of bias will be 0.003 * size of texel, to reduce the self-shadowing. This value was based on multiple tuning to achieve the best result. Finally, the coordinates can be used to sample from the shadow mask.

1.7 Result

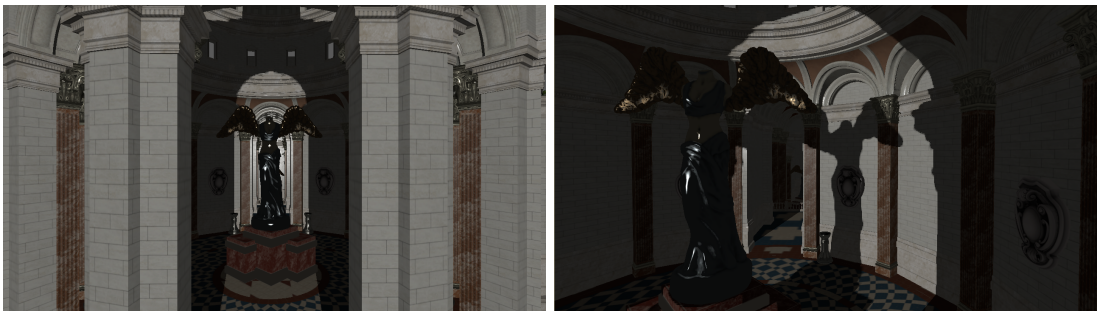


Figure 1.2: result of shadow mapping

Figure 1.2 shows the result of shadow mapping in different angles with PCF applied. By adjusting the light position, the shadow will move along with the light.

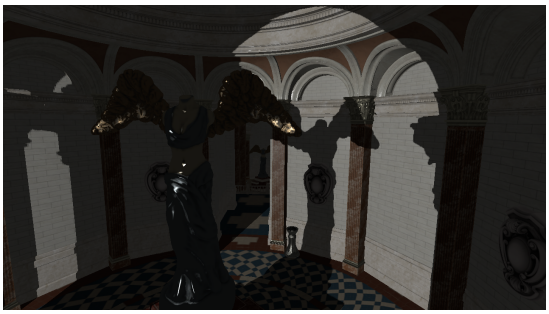
1.8 Comparisons

Figure 1.3: with PCF

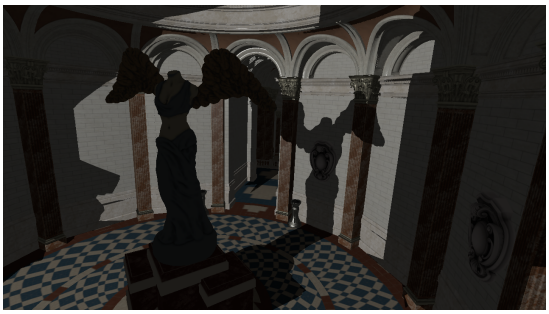
Figure 1.4: without PCF

Figure 1.5: PCF

Figure 1.5 compares the result applied PCF and without applying PCF. It clearly shows that after applied pcf, the shadow will be more smooth. My method for applying 2x2 PCF is in the fragment shader, when sampling from the shadow mask, instead of only sample one value, I took the 4 surrounding values in consideration and took the average value.

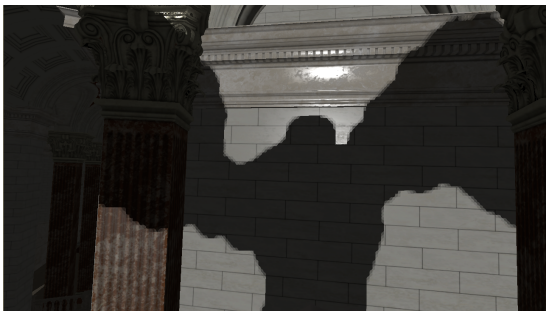


2048x2048

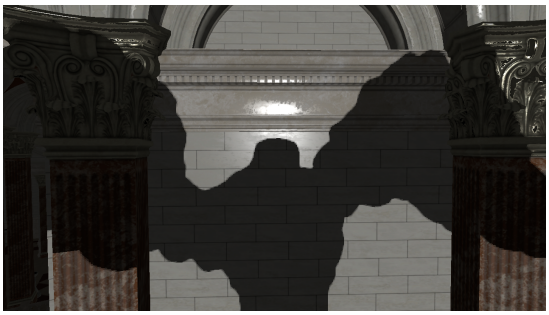


4096x4096

Figure 1.6: Different resolution



2048x2048



4096x4096

Figure 1.7: Different resolution PCF result

Figure 1.6 and 1.7 compares the result of different resolution (2048x2048 and 4096x4096). The results will have no significant difference from a distance. However, if taking a close look, it suggests the higher the resolution, the sharper the PCF effect will have.