UNIVERSITY OF LEEDS

Assignment 2 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

April 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.

Contents

Chapter 1

Task 1 - Prep and Debug

Overall, the preparation and Debug set up follows the footsteps of the previous assignment 1. The implementation of the renderer mainly referenced the procedure in assignment 1 and the exercise tutorial handbook.

The project will first initialise a Vulkan Context, in this case a VulkanWindow object, with a renderable GLFW window handle correctly setup. Then, the crucial settings, listed as follows, will be correctly implemented:

1. Instance layers and extensions
2. Enabled layers and extensions
3. Logical device
4. GraphicQueue for rendering
5. Swapchain
6. Framebuffers for the swap chain images
7. Render pass, pipeline layout
8. Pipeline layout
9. Basic command queue and function for submitting command
10. A frame-rate independent camera movement implementation

The main changes compared with the previous coursework are in the process of handling mesh

data, creating buffers, shader and rendering commands.

For the method I chose for handling the mesh data, I had created a structure storing all the useful data in `vertex_data.hpp`, namely `TextureFragment`. This structure will hold four Buffers for vertex positions, texture coordinates and indices for now. (vertex normals and tangents will be added in task 2 and 3 respectively). Moreover, the IDs of the textures, including base colour, roughness and metalness will also be recorded as `std::uint_32`. (as before, the normal mask and alpha mask will be add in the later implementation.) Finally, the count of indices will also be passed for the purpose of calling the `vkCmdDrawIndexed` function. A `TextureFragment` object can be achieved by calling the function `handle_baked_model`. Another helper function `getImages` will load all the texture will be used for rendering, by reading the path provided by the `BakedModel` and then store their image and image view in the two vector, namely `imageVec` and `imageViewVec`. With all the data prepared, the meshes are ready to be rendered.

To achieve the 3 results the first task is asking to implement, I had created 3 different shaders with their own rendermode, which can be switch by pressing the number button on the keyboard. The image of rendering effect will be list as follows:

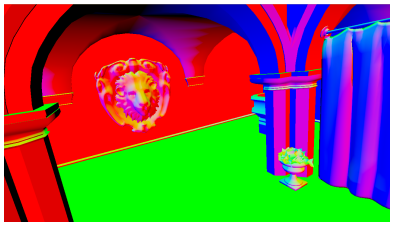

Figure 1.1: Normal

Figure 1.2: View direction

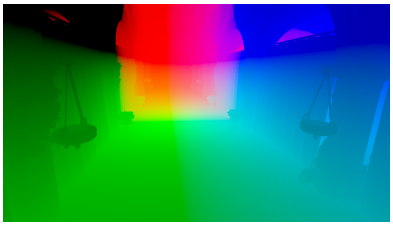

Figure 1.3: Light direction

Rendering of vertex surface normals can be achieved by passing the normals to the vertex shader, and send it as the colour to the fragment shader.

Rendering of view direction can be achieved by acquiring the distance between the position of the camera and the vertex in shading space. Inside the uniform buffer, a `UScene` contains a camera matrix, which can be used to calculate the camera position by take the transform vector. Then with this position and the vertex position together pass to the fragment shader, the result can be calculated.

The implementation of rendering the light position is similar as the method for rendering of view direction. The position of light in the shading space was sent to the uniform buffer and can be acquired by the vertex shader. The vertex colour can be calculated as the light position

minus the vertex position.

For now, the descriptor layout has only one binding, the base colour. The reason why this binding is necessary is for checking if the mesh data is correctly set up, and the rendering result is shown as figure 1.4. By comparing with the result from coursework 1, the result suggest the mesh data and the rendering preforms correctly.

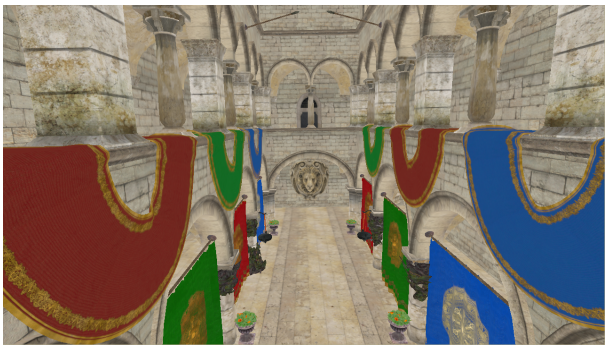

Figure 1.4: base colour

The uniform input data includes a `UScene` structure, which includes the camera matrix, projectction matrix and light position in shading space (0, 2 ,0).

Chapter 2

Lighting

Figure 2.1: D

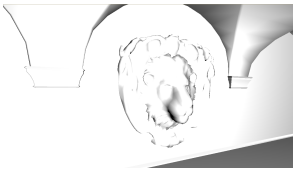

Figure 2.2: G

Figure 2.3: F

Figure 2.4: PBR specular term

To achieve this task, I had created a pair of vertex shader and fragment shader, namely `lightTex.vert` and `lightTex.frag`. The lighting model strictly follows the PBR model given by the coursework requirement, which introduced three factors.

Figure 2.5: physical properties contribution

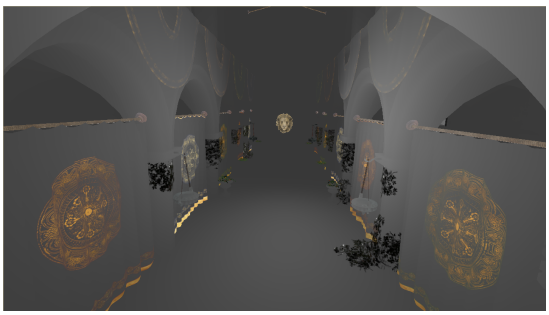

Figure 2.6: PBR specular term

Figure 2.1 show the normal distribution term of the lighting model, the term describe the small variations in the object's geometry. The term is calculated by mainly considering the physical properties of the material, and the position relationship between the object, light and the camera. By observing figure 2.5, it can be concluded that with different physical properties, the curtain and the stone column reflect different among of light. Since the cloth has rougher

material than cobbles(my assumption of the texture of the stone column), also considering cobbles has higher metalness, the stone column should perform more like a smooth metal than curtain. Therefore, some parts of the stone column reflect more light than the curtain, whereas reflection from other parts could hardly be observed. On the other hand, Figure 2.1 suggests that although sharing the same material on the wall, vertices has smaller angle between normal and half-angle reflect more light, this matches the specular reflection effect.

The rendering result of mask term is shown in Figure 2.2. The mask term, G, describes the self-shadowing where microfacets block each other (UniversityofLeeds 2023). My expectation is that on the same object, the parts that are blocked by other parts from being illuminated by the light or reflecting light to the camera should has darker colour. Figure 2.2 suggest the same result as my guessing, where the vertex behind the underjaw of the lion has darker colour compared with other area.

The result of Fresnel term F is shown in Figure 2.3, describing the amount of specular reflection. As the primary contributor to the Fresnel term is the metalness of the texture, objects have higher metalness are expected to have the colour more similar to its original, by contrast, objects has lower metalness should have the colour more similar to the approximation value(0.04, 0.04, 0.04). Figure 2.6 suggests the result meets the expectation (the rendering of the plants can be ignored). Figure 2.6 also suggests the position relation contribute to the Fresnel term, by observing the different colour of the structure of building.

The light source is programmed to be animated and can be played with by the user. By pressing `i, j, k, l` the user can control the horizontal transform of the light position, pressing `u, o` the user can lift or sink the light source. The program also allows the user to hold pressing space bar to rotate the light source along the y-axis.

Chapter 3

Task 3 - Alpha Masking

To achieve this task, I had created a new render pipeline, namely alpha pipeline. All meshes that are specified having alpha term will be rendered in this render pipeline. With correctly setting the binds and the `blendInfo`, as well as adding the corresponding commands, figure3.1 shows the result on meshes with alpha term. If we compare figure3.1 to figure3.2, it is obvious that the textures of plants are cleaner and all the unnecessary parts were masked.

Figure 3.1: alpha pipeline applied

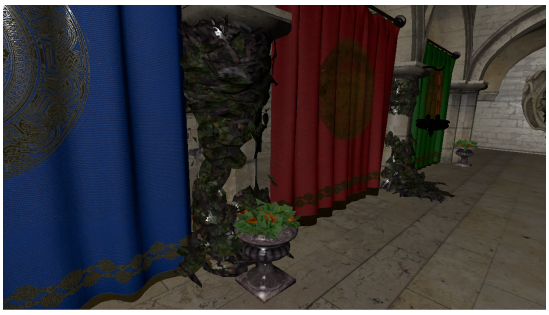

Figure 3.2: without alpha pipeline applied

Chapter 4

Task 4 - Normal Mapping

To achieve normal mapping, I had divided the task into 3 goals by reading the requirements: to calculate tangent vector, to read/write the newly calculated data from/into the binary file, and to calculate the new normal for the BRDF lighting calculation.

The first goal was achieved by using the `tgen` library. By modifying the code in `index_mesh.cpp/hpp`, I had called all four functions provided by `tgen` library in the correct order to generated the tgen vector. Then, the tangents will be stored in the `IndexedMesh` structure as a vector of `glm::vec4`, noting the last component of the tangent vector represents if the vector needs to be flipped for avoiding explicit binormals.

Then, in `cw2-bake/main.cpp`, codes for writing the tangents into the binary file was added. Correspondingly, in `cw2/baked_model.cpp`, the tangents data will be read and stored in the `BakedMeshData` structure. Ultimately, the tangents data will be sent to the rendering pipeline through buffers.

Finally, in the fragment shader, the final normal that will be used in lighting calculation will be calculated as follows:

vec3 finalNormal = normalize(mat_TBN * oNormalSamp);

, where `oNormalSamp`stands for the normal sampled from the normal mask, `mat_TBN` is the TBN matrix, calculated as:

mat3 mat_TBN = mat3(oTangent, biTangent, surfaceNormal); // TBN mat

here the `oTangent` stands for the normalised tangent vector, `biTangent` can be calculated as:

vec3 biTangent = normalize(v2fTangent.w * cross(v2fNormal, oTangent)); // bi-Tangent

, where `v2fTangent` is the tangent vector, `v2fNormal` is the original surface normal.

The reason why `v2fTangent.w` is necessary is this value records the direction of biTangent. Having this the forth components inside the tangent vector means the programmer does not need to expensively pass biTangent vectors together with the tangents, instead, they can be achieved by the previously mentioned formula.

Replace the original surface normal with the calculated final normal, each vertex will now have its own normal, instead of sharing the normal with other vertex on the same facet. By comparing the effect shows in figure4.1 and figure4.2, it is suggested that implementing normal mapping will give more details to the small facet in the lighting and shadowing. This can also be observed by comparing their normal visualisation in fugure4.3 and figure4.4.

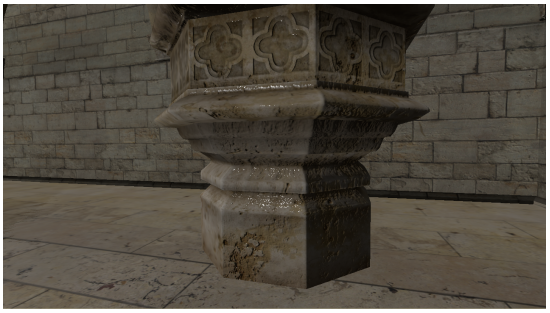

Figure 4.1: normal mapping applied

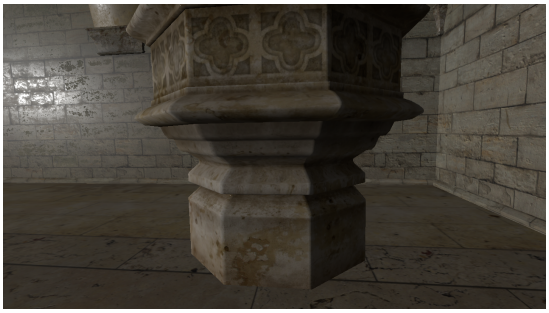

Figure 4.2: without normal mapping applied

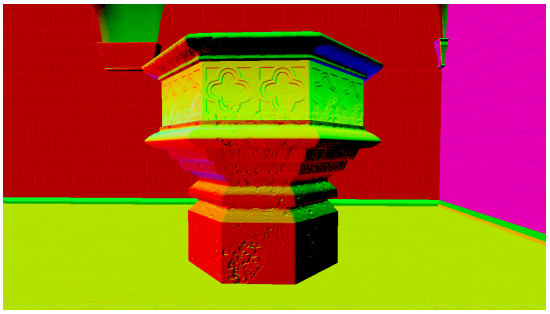

Figure 4.3: final normal

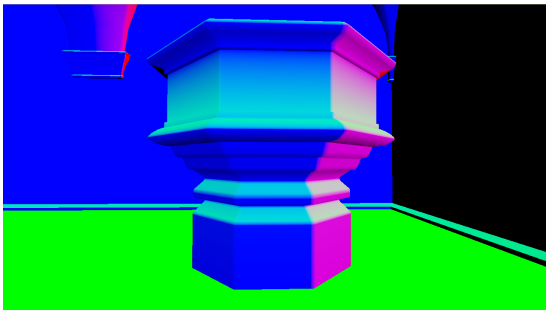

Figure 4.4: surface normal

References

UniversityofLeeds (2023). “COMP5822M Coursework 2”. In: *COMP5822M Coursework 2*, p. 5.