UNIVERSITY OF LEEDS

Assignment 3 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

1 Task 1 - Render to texture **1**

2 Task 2 - Tone Mapping **4**

3 Task 3 - Bloom **5**

Chapter 1

Task 1 - Render to texture

To implement this task, my general idea is: to create 2 render passes, first one will be used to render off-screen intermediate images, and the other render pass will be used to gather all the images and render to the swapchain framebuffer.

Nevertheless, before that, I had implemented a single-pass renderer without render to texture method, just for make sure the models were correctly loaded. The structure of the models is slightly different from the models were given in coursework 2, instead of having identical base colour, metalness and roughness for individual vertices, vertices in each mesh will share common values of the three previously mentioned attributes. One way of implement could be store the base colour, metalness and roughness N times in separate buffers, where N equals to the number of vertices in the mesh. However, this method could consume vast of memory for storing the redundant values, also would asked the GPU to do a lot more computation. This method is clearly labour consuming and inefficient. Another way could be store these value in a structure and upload the structure in an uniform buffer. This method will save a lot of memory and it only asks for a updating in the buffer. However, this method requires vkFences to guarantee synchronisation. The way I chose to solve this problem is by using `vkCmdPushConstants` function. This is a very cheap way of pushing a small amount of data to the shaders during rendering time. With correctly setting up a structure and adding extra code in the pipeline layout, I could call `vkCmdPushConstants` in record_commands function. Each mesh will call the function once to push its own data to the shader, and the data will be used to calculate intermediate images.

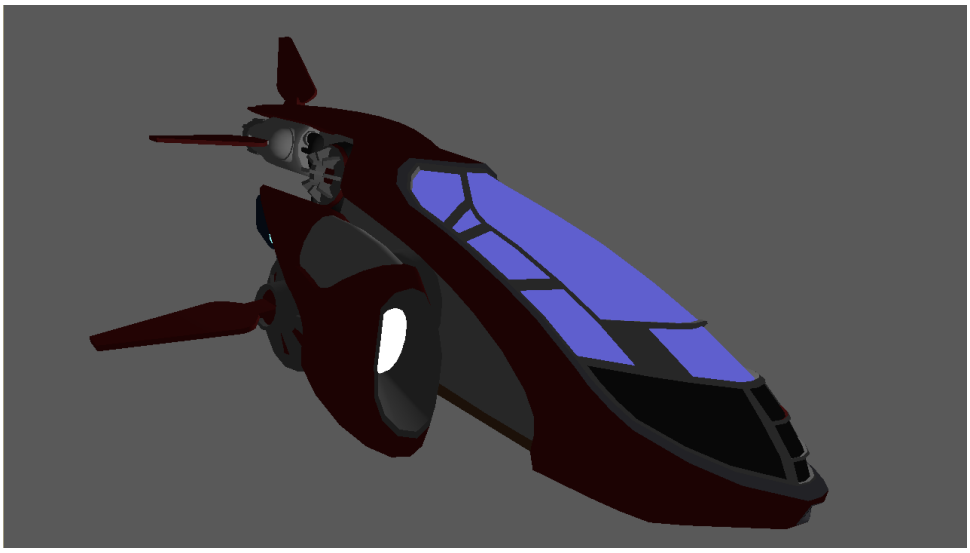

Figure 1.1: single-pass render result

Figure 1.1 shows the result of a successful implementation, with same PBR calculation as the one used in coursework 2.

After that, I had started to create separate render passes. For task 1, the first render pass will take in two attachments: one has format`VK_FORMAT_B8G8R8A8_SRGB` for storing the base colour, the other ha format `VK_FORMAT_D32_SFLOAT` will be served as depth buffer; the output will be an image storing base colour data. Each attachment has its own imageview handle, which owns a VkImage for storing data. The difference between the colour attachment in cw3 and cw2 is that in cw3, intermediate images will have format of shader read only, instead of present mode. Then, in the second render pass, it will take in the colour image and output colour in the swapchain images. The result is shown in figure 1.2.

Since I only created one command buffer, it does not need synchronisation within, however, the updating of scene uniform buffer does need a fence to wait it finishes its work. I had add two dependencies for both passes. Both were set to an external source to subpass 0 before the pass, and from subpass 0 to external source.

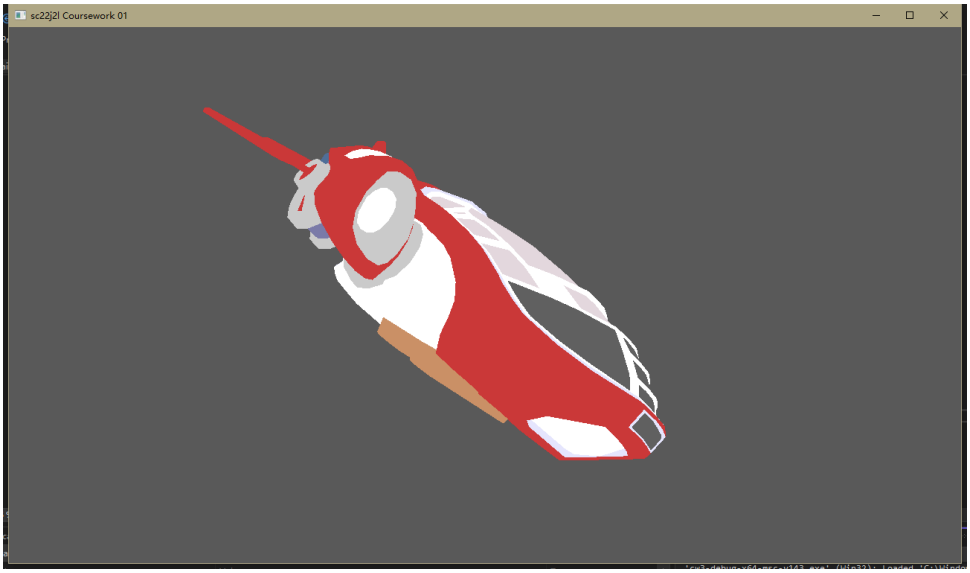

Figure 1.2: 2 passes render result

Chapter 2

Task 2 - Tone Mapping

To implement tone mapping, I had modified the output colour by implementing the given formula in fragment shader in the full-screen render pass.

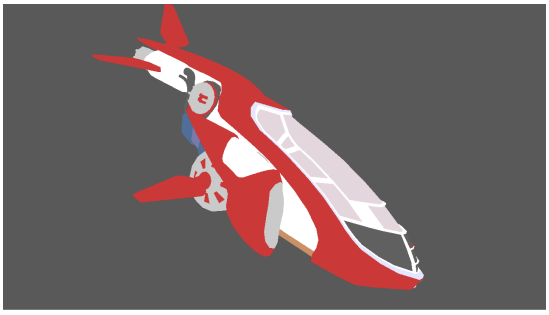

Figure 2.1: without tone mapping

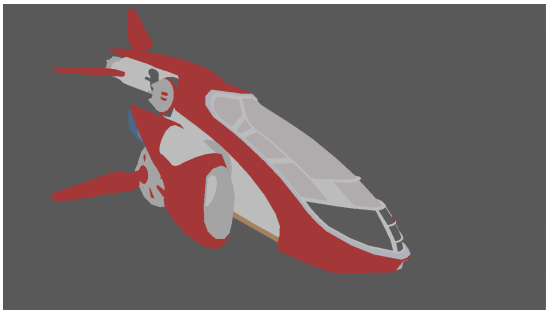

Figure 2.2: with tone mapping

By comparing figure 2.1 and 2.2, it can be concluded that with implementing tone mapping, the colour got a little bit darker.

Chapter 3

Task 3 - Bloom

To implement this task, I had applied significant changes in the previous code. First, I had add a PBR render pass between the compute pass and the full-screen pass, for calculating lighting effects. Now, the compute pass will take six attachments, recording: vertex base colour (`VK_FORMAT_B8G8R8A8_SRGB`), vertex positions (`VK_FORMAT_R32G32B32A32_SFLOAT`), vertex normals (`VK_FORMAT_R32G32B32A32_SFLOAT`), vertex material (`VK_FORMAT_B8G8R8A8_SRGB`) (roughness and metalness), emissive colour (`VK_FORMAT_B8G8R8A8_SRGB`), and depth. The first five of them will be passed to the PBR render pass, for calculating lighting effect. The result is shown in figure3.1.

Figure 3.1: single-pass render result

The PBR render pass its own will have two attachments: one for recording the calculated colour, and the other will record the filter value. All vertex with either R, G or B value greater than

1,0f will be considered as emitting light, and will be recorded for the late calculation. Figure 3.2 shows an example of the filter image.

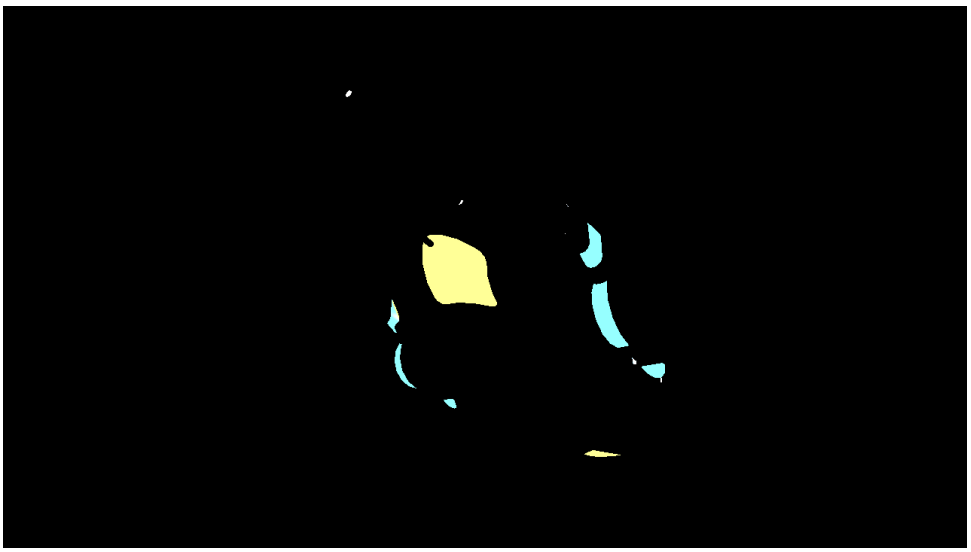

Figure 3.2: single-pass render result

Due to lack of time and vast debugging in previous stage, I could not finish this task. The next procedure should be creating another 2 render passes for applying the Gaussian blur horizontally and vertically with the given weight. By correctly implement that, in the full-screen pass could blend the Gaussian blurred image with the PBR colour image, thereby some halo effects could be obsvered.