UNIVERSITY OF LEEDS

Assignment 1 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

March 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 - Vulkan infrastructure

1.1 Creating a Vulkan instance

To Initialise a Vulkan instance, several parameters needs to be correctly set up.

`VulkanContext.cpp/hpp` are responsible for the implementation for initialising a Vulkan objects and setting up its relevant parameters. The code was done by referencing the exercising handbooks, hence my own understanding on the code will be illustrated in the following content.

When `make_vulkan_context` (in `VulkanContext.cpp`) is called, the function will create a Vulkan instance following these steps:

1. Initialise Volk
2. Check for instance layers and extensions
3. Create Vulkan instance with the enabled layers and extensions
4. Create a logical device
5. Retrieve the graphicQueue for rendering
6. Return the newly created vulkan context

Step 1 is achieved by calling function `volkInitialize()`, facilitating the usage of the Vulkan API. Step 2, 3 and 4 will be further analysed in the later corresponding sections.

It is worth noting that the `VulkanContext` objects are designed to be move-only, instead of copy-able. The reason for this is avoiding the confusion and errors in resource management.

Each VulkanContext object contains its own resources, and only itself has the right to create and destroy these resources. Setting copying a VulkanContext object to be legal may lead to a circumstance that multiple VulkanContexts try to manage the same resource and end up with incorrect usage and destruction of resources. This is achieved by deleting the copy constructor and copy assignment operator in `VulkanContext.cpp`:

```
VulkanContext( VulkanContext const& ) = delete;
VulkanContext& operator= (VulkanContext const&) = delete;
```

In Step 5, `vkGetDeviceQueue()` function is called to pass the location of the graphicQueue to `ret.graphicsQueue`, which is a parameter of the VulkanContext object. My machine has 4 Queue Families as shown in the followings:

```
Queue family: QUEUE_GRAPHICS_BIT | QUEUE_COMPUTE_BIT | QUEUE_TRANSFER_BIT
                | QUEUE_SPARSE_BINDING_BIT (16 queues)
Queue family: QUEUE_TRANSFER_BIT | QUEUE_SPARSE_BINDING_BIT (2 queues)
Queue family: QUEUE_COMPUTE_BIT | QUEUE_TRANSFER_BIT |
                QUEUE_SPARSE_BINDING_BIT (8 queues)
Queue family: QUEUE_TRANSFER_BIT | QUEUE_SPARSE_BINDING_BIT |
                VkQueueFlags(100) (1 queues)
```

where the graphicQueue is a member of the first Queue Family, who owns 16 queues.

This class provides a lot of useful base functions and will be inherited and extended by the VulkanWindow class, which will help to build a renderable window. This part will be explained in Section 1.4.

1.2 Enabling validation layers in debug builds

This section is related to the step 2 mentioned in Section 1. To enable layers and extensions, The code first enumerate instance layers and extensions, and then store them into lists, respectively. This is achieved by calling the two functions defined in `context_helpers.cpp` :

```
std::unordered_set<std::string> get_instance_layers();
std::unordered_set<std::string> get_instance_extensions();
```

The two functions above inside called the following two functions from Vulkan API:

```
vKEnumerateInstanceLayerProperties()
vKEnumerateInstanceExtensionProperties()
```

each API function were called twice, for first enumerating the layers or extensions, and then return them into a list.

Here I will note a confusion while I was studying the code provided in the exercise handbook. At first, I used to think `vKEnumerateInstanceLayerProperties()` this function was enumerating the layers that the device is supported, after reading the `khronos.org` tutorial and searching on google I found out that as claimed in the name, it would enumerating the layers of instance but not device. One may call `vkEnumerateDeviceLayerProperties()` to enumerate the layers that devices support. Also noting that return result of the latter function has an extra pointer to a `VkLayerProperties` structure, describing the amount, names, version and description of the layers that the devices support.

1.3 Creating a renderable window

The code for this part follows the tutorial in the exercise handbook, using GLFW to initialise a renderable window. The `VulkanWindow` class inherits the `VulkanContext` class, daclared in the `vulkan_window.hpp/cpp`. The class extends the original VulkanContext class, owning a GLFW handle and some other necessary parameters. The function `make_vulkan_window` will be responsible for the creation of a renderable window. Once the function is called, it will first check the initialisation of the GLFW and whether it supports Vulkan, and then check the availability of the necessary extensions and enable them, finally create the window with given parameters. The VulkanWindow will also be move-only, since it inherit the VulkanContext class.

The layers and extensions that my program enables are:

```
Enabling layer: VK_LAYER_KHRONOS_validation
Enabling instance extension: VK_KHR_surface
Enabling instance extension: VK_KHR_win32_surface
Enabling instance extension: VK_EXT_debug_utils
```

```
Enabling device extension: VK_KHR_swapchain
```

1.4 Selecting and creating a Vulkan logical device

The construction of the code for selecting and creating a Vulkan logical device follows the tutorial in the exercise. Firstly, inside `select_device` function, `vkEnumeratePhysicalDevices` was called twice to enumerate the physical devices and store them in a list. Then, `score_device` will be called to mark a score to each physical device though checking the availability of swap chain extension and a valid queue family. The physical device has the highest mark will be pass to `create_device` function, along with the necessary queue families and the enabled device extensions, for creating a logical device.

The loader on my machine has the Vulkan version: `1.3.224`.

The selected and the only device on my machine is `NVIDIA GeForce RTX 3060 Ti`, which has the property: `(Vulkan: 1.3.224, Driver: 526.98.0.0)`, and type: Type: `PHYSICAL_DEVICE_TYPE_DISCRETE_GPU`. This device supports Anisotropic filtering for task 3.

1.5 Creating a swap chain

The construction of the code for creating a swap chain follows the tutorial in the exercise. The `create_swapchain` function is responsible for creating a swap chain. It will first enumerate and record the format and the presentation mode the device supports, by calling the two functions: `get_surface_formats` and `get_present_modes`. The code will then primarily select the format `VK_FORMAT_R8G8B8A8_SRGB` or `VK_FORMAT_B8G8R8A8_SRGB` if the device support either of them. The first device-supported format was stored in the list will be selected if both of the previously mentioned formats are not supported. Then, the present mode was set to `VK_PRESENT_MODE_FIFO_KHR`, as required in the question. After that, the code inquire the range image count to decide the image count number will be used in the program. The final image count selected on my machine is 3. Finally, with the parameter correctly passed in, the function can build a `VkSwapchainCreateInfoKHR` to create a swap chain.

1.6 Creating framebuffers for the swap chain images

This part follows the tutorial in the exercise. In `vulkan_window.hpp`, function `get_swapchain_images` will get the swapchain image handles and then they will be passed to `create_swapchain_image_views`, where the ImageView will be created.

1.7 Creating a render pass

This part follows the tutorial in the exercise. The render pass is created inside the function `create_render_pass`, locates in `main.cpp`. The render pass will have 2 attachments, one for the swap chain image, the other one is for the depth buffer. The render pass will have only one sub-pass and two dependencies. Then with the `VkRenderPassCreateInfo` correctly setup, a render pass can be created.

1.8 Repeatedly recording commands into a command buffer and submitting the commands for execution

This part mostly follows the tutorial in the exercise. In order to record commands, a command pool and command buffer need to be created first. These two objects will be created by the two functions in `vkutil.cpp`, called `create_command_pool` and `alloc_command_buffer`, with correctly setting `poolInfo` and `cbufInfo`.

Then, after all the data for the current frame is ready, commands will be recorded in `record_commands`. After that the recorded commands will be submitted in `submit_commands`. Two semaphores, namely `imageAvailablewill` and `renderFinished` were created to guarantee that the rendering commands will be executed after the swapchain image is available, and the presentation of the swap chain image will happen after the rendering commands finished.

Chapter 2

Task 2 - 3D Scene and Navigation

2.1 Rendering the object

To achieve this task, first, in `vertex_data.hpp` the .obj file will be loaded as a SimpleModel object using the provided obj-loading function. Then, two functions, namely `handle_textured_mesh` and `handle_untextured_mesh`, were defined to process the SimpleModel, partially referenced the tutorial in the exercise 1.4 and 1.5. The first function will focus the textured meshes, push all the vertex locations and texture coordinates into the created vertex buffer, record the offset, vertex count and the index of the texture for each mesh, and then return the result in a `TexturedMesh` structure. The latter function will do similar thing but focusing on the untextured meshes and return a `ColorizedMesh`. The second function will record material diffuse colour instead of texture coordinates, and it does not have to record the texture index. All the buffers in the two function will be protected by buffer barriers, so that there will be guaranteed no synchronisation issue while accessing these buffers.

At the mean while, two render pipelines will be created for rendering textured and un-textured meshes, respectively. The creation of pipeline layout referenced the tutorial in the exercises, and the creation of the pipelines were based on the needs of the rendering.

Then, in `main.cpp`, before falls into the main loop, the texture images need to be loaded for the rendering process. A list of the paths to the textures (.png images) will be generated by calling the function `getTexPath` in `vertex_data.cpp`. The paths will be used to locate textures and create ImageViews. Then, these ImageViews will be used to create VkDescriptorSet. The

generated VkDescriptorSet will be stored in a vector, for matching the textured mesh while rendering.

Finally, with all the data ready, rendering commands for both pipeline will be recorded in `record_commands`. For the textured meshes pipeline, the descriptors will be bind follows the index of textures generated by the function `handle_untextured_mesh` in `vertex_data.hpp`, and for both pipeline the vertices will be drawn follow the offset and vertex count for each mesh by calling `vkCmdDraw`. figure 2.2 and 2.2 shows the results of rendering.

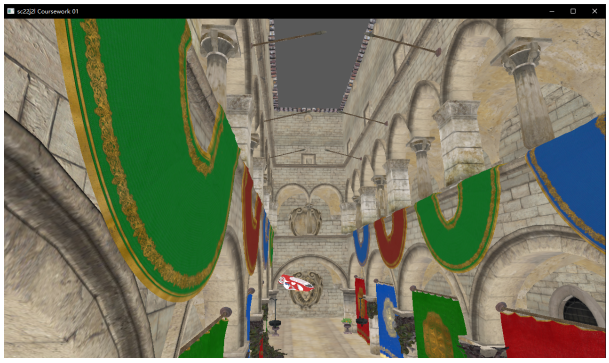

Figure 2.1: overview rendering result

Figure 2.2: close-up of the un-textured meshes

Overall, the logic of my implementation can be briefly concluded as follows: during the assets loading procedure, the .obj and .mtl files will be read in as a SimpleModel object. The vertex attributes will be pushed into buffers. Important indices will be stored with each mesh. The .png textures images will be loaded in and used to create descriptors. Two pipelines will be created for rendering textured and un-textured meshes, respectively. Then, for each frame, each mesh will call the `vkCmdDraw` function once to draw itself, provided its offset and vertex count.

2.2 Camera

This part follows the tutorial in the exercise 1.5. Several GLFW call back were implemented to receive the user inputs, these includes:

1. `glfw_callback_key_press()`: for checking which key did user press on the keyboard and trigger relevant event.
2. `glfw_callback_button()`: for checking which mouse button did user click, if right button was clicked, enable or unable the rotation of the camera.
3. `glfw_callback_motion()`: for recording current cursor position, this helps calculating the

rotation of the camera.

For each frame, the user state, including the camera orientation, camera position will be updated before rendering process. The position and the orientation of the camera will be calculated and sent to shader through a uniform buffer. This unifor mbuffer will be created by calling the fuction `create_buffer` defines in `vkutil.cpp` with setting the flag to be `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT`. Then this buffer will be attached into the scene descriptor.

Chapter 3

Task 3 - Anisotropic filtering

To achieve this task, I modified several place in my code. The first change is to enable Anisotropic filtering when creating the logical Vulkan device. In line 741 in `vulkan_window.cpp`, I added this line:

```
deviceFeatures.samplerAnisotropy = VK_TRUE;
```

The second change is line 735 to line 738:

```
vkGetPhysicalDeviceProperties2(aPhysicalDev, &properties);
auto limits = properties.properties.limits;
float maxAnisotropy = limits.maxSamplerAnisotropy;
std::printf("The supported maximum sampler anisotropy
            value is:  %.2f\n", maxAnisotropy);
```

This piece of code is to print the limits that the Vulkan implementation defines. The printed result is 16.00.

Therefore, in the final change in line 294 and 295 in `vkutil.cpp`, I added these:

```
samplerInfo.anisotropyEnable = VK_TRUE;
samplerInfo.maxAnisotropy = 16.0f;
```

The number 16 was chosen to make the difference as obvious as possible.

The following figures shows the difference between Anisotropic filtering turned on and off:

Figure 3.1: Anisotropic filtering on

Figure 3.2: Anisotropic filtering off

It is clearly that when the camera is close to the floor, while Anisotropic filtering turned on, the texture of the floor far from the camera is clearer than Anisotropic filtering turned off.

Chapter 4

Task 4 - Visualizing Mipmaps

To implement this part, I have made 2 primary changes: updating the fragment shader to render the mipmap level and modifying the user state related code to allow user to switch between different render modes.

First, in the new fragment shader, namely `shaderMPMP.frag`, I added this line in main() to inquire the mipmap level at current pixel:

```
float level = textureQueryLod(tex, texCoord).x;
```

I found this function by searching on the google, and this *website* 2018 came out.

Then I recorded 9 different colours: black, red, orange, yellow, green, cyan, blue, purple and white to encode the level of detail. The pixel will be rendered with the corresponding colour, the colour towards to the right of the queue represents a higher level of detail.

Inside `main.cpp`, I added two new member to the `EInputState` enumerator, representing the normal texture rendering mode and the mipmap level colour mode. Then, I added the `changeMode` boolean value inside the `UserState` structure. Furthermore, inside cfg namespace, I cerated an enumerator containing the two render mode flag, and a global variable `currMode`, which is set to be `RENDER_MODE_TEXTURE` by default. Finally, the `update_user_state` and `glfw_callback_key_press` function was modified to match the newly add functions. If the user press "1" or "2" on the keyboard,which representing switching rendering mode, the event will be triggered. At the beginning of the main loop, the program will check if the `changeMode` boolean value inside the `UserState` structure is set to be true, and if so, the program will

manage to recreate the swap chain and render pipelines, so that the pipeline will choose the correct fragment shader between the old and the new.

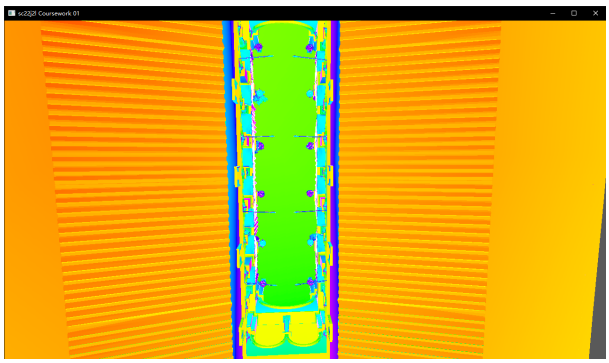

Figure 4.1: far distance

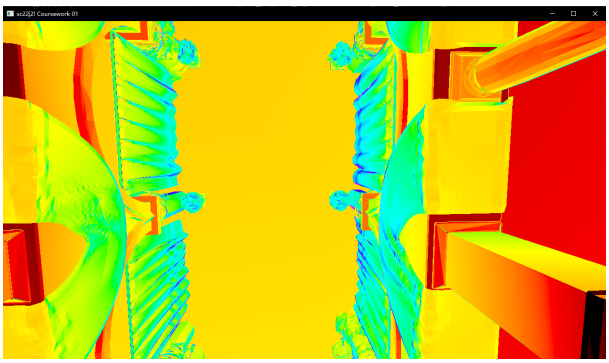

Figure 4.2: near distance

By observing figure 4.2, the colour suggests the highest level is 11-12, represented by white. This suggests that the highest mipmap level, as will be printed in the cmd window, was used at the render process. However, if the zoom in on the object, the object tends to use lower mipmap levels.

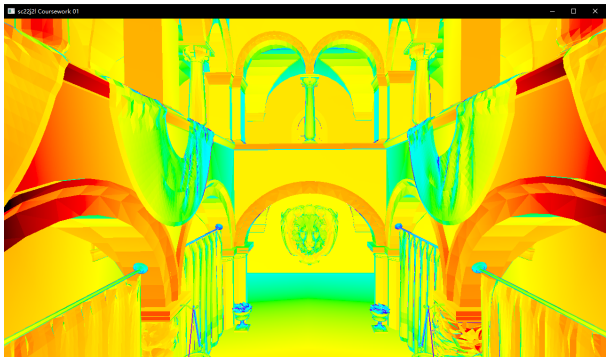

Figure 4.3: window width=1280, height=720

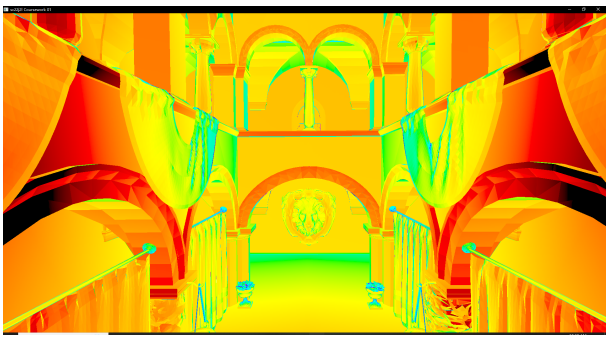

Figure 4.4: window full-screen

Figure 4.3 and figure 4.4 shows the same scene with different window size. Comparing the two figure, it can be concluded that changing the size of the window may affect the calculation of the level of details (see the colour on the line face in the middle of the picutre). Therefore, it can be predicted that if a higher resolution is applied, there may be more levels of detail and the calculation of level may be affected. This is suggesting that one can reduce the resolution or decrease the size of the window to increase efficiency of the rendering.

References

website (2018). Ubuntu. URL: https://manpages.ubuntu.com/manpages/impish/man3/textureQueryLod.3G.html (visited on 03/23/2023).