COMP5822M Coursework 1

Contents

In Coursework 1, you will be asked to write a small Vulkan-based renderer. CW 1 utilizes the material covered by the first four exercises. In CW 1, you will:

- Implement the basic Vulkan rendering infrastructure.
- Load and render multiple objects.
- Load and use textures, utilizing mipmapping and anisotropic filtering.
- Implement a user-controlled camera.

Before starting work on the CW, make sure to study this document in its entirety and plan your work. Pay in particular attention to Section 2, which contains information about submission and marking.

If you have not completed Exercises 1.1 through 1.4, it is heavily recommended that you do so before attacking CW 1. When requesting support for CW 1, it is assumed that you are familiar with the material demonstrated in the exercises! As noted in the module's introduction, you are allowed to re-use any code that ***you have written yourself*** *for the exercises in the CW submission. It is expected that you understand all code that you hand in, and are able to explain its purpose and function when asked.*

While you are encouraged to use version control software/source code management software (such as git or subversion), you must ***not*** *make your solutions publicly available. In particular, if you wish to use Github, you must use a private repository. You should be the only user with access to that repository.*

You are encouraged to build on the code provided with the coursework. If you have completed the exercises, you will already be familiar with the code and have some of the missing parts. If you wish to "roll your own", carefully read the additional instructions in Appendix A.

CW1 uses the tinyobjloader library to load 3D models in the Wavefront OBJ format. To spare you some frustrations when dealing with some of the more obscure "features" of the Wavefront OBJ format, CW1 includes the `cw1/model.hpp` and `model.cpp` sources, which define a few structures and the `load_obj_model` function. You may use the latter to load a specific OBJ file into CPU memory.

CW1 uses an older version of the *tinyobjloader* library, version 1.0.6. This version has been battle-tested for a few years.

1 Tasks

The total achievable score for CW 1 is **30 marks**. CW 1 is split into the tasks described in Sections 1.1 to 1.5. Each section lists the maximum marks for the corresponding task.

Do not forget to check your application with the Vulkan validation enabled. Incorrect Vulkan usage -even if the application runs otherwise- may result in deductions. Do not forget to also check for synchronization errors via the Vulkan configurator tool discussed in the lectures and exercises.

General submission quality

4 marks

Up to **4 marks** are awarded based on general submission quality. Submission quality will be determined from several factors. Examples include overall readability/structure of your code (e.g., consistent style, naming and commenting), good coding practices, and how easy it is to build your code. Additionally, carefully read Section 2. Minor errors with respect to the instructions there may result in deductions of submission quality.

Your submission should solve the tasks presented in the coursework. Inclusion of large amounts of unnecessary code may result in deductions.

1.1 Vulkan infrastructure

8 marks

Start by setting up the necessary Vulkan rendering infrastructure for a real-time rendering application. This likely includes:

- Creating a Vulkan instance
- Enabling the Vulkan validation layers
- Creating a renderable window
- Selecting and creating a Vulkan logical device
- Creating a swap chain
- Creating framebuffers for the swap chain images†
- Creating a render pass†
- Repeatedly recording commands into a command buffer and submitting the command buffer for execution
- Performing necessary synchronization

†You are allowed to use Vulkan 1.2 or Vulkan 1.3 functionality or equivalent extensions (imageless framebuffers or dynamic rendering) instead. In this case these two steps change slightly. If you decide to go down this route, make sure to mention this in the README with your submission (Section 2).

Your application must allow the window to be resized and handle the resizing appropriately (i.e., it must recreate the swap chain and related resources when necessary).

If you can draw anything you specify, including a solid color, on the screen, you have likely completed this task.

1.2 3D Scene

7 marks

CW1 includes two Wavefront OBJ files, `car.obj` and `city.obj`, along with material definitions (`*.mtl`).

Extend your application to load these *two* OBJ files, and render a 3D scene showing objects from both files. You should not change the provided OBJ files (or their material definitions).

Each object consists of one or more meshes that each have a different material (see `cw1/model.hpp` and `cw1/model.cpp`). Hence, you will need to deal with multiple materials. One simple way is to reserve a descriptor set for material information. Then, during the loading phase of your application, create a separate uniform buffer and/or texture and `VkDescriptorSet` instance for each material. When drawing, you will just need to bind the correct descriptor set instance before drawing a certain mesh (`vkCmdDraw`). (A specific "gotcha" is that Vulkan does not allow the use of `vkCmdUpdateBuffer` inside a render pass, meaning you cannot use it to easily update a single uniform buffer with new material information on the fly.)

Take special care regarding the materials. Some materials contain textures, while others use solid colors only. You will need to handle both cases. There are several options for this, including creating two separate graphics pipelines (one for each case), or creating a 1×1 "dummy" texture for meshes with only solid colors.

Using a `if` statement to choose between two code paths in the fragment shader may be tempting, but is likely a suboptimal choice.

You do not have to implement any lighting. Do enable back face culling and depth testing. Your results should look similar to the teaser image.

The textures are provided in the JPEG format (`.jpg`). The `stb_image.h` library introduced in the exercises can load JPEG files as well as PNGs.

1.3 User camera

4 marks

Extend the application to include a user-controllable camera. The camera should be controlled with the keyboard and mouse.

Control position with the WSAD and EQ keys:

- `W` - forward
- `S` - backward
- `A` - move/strafe left
- `D` - move/strafe right
- `E` - move up
- `Q` - move down

(directions relative to the camera's orientation).

Movement speed should be increased when holding the Shift-key, and decreased when holding the Ctrl-key.

Use the mouse to control the camera's orientation. Aim for a "first person camera". Mouse navigation should be activated when the right mouse button is clicked and deactivated again when it is clicked a second time.

Both movement speed and camera rotation rate should be independent of the frame rate.

In GLFW, look at the Input guide and Input reference pages. In particular, look for

- `glfwSetKeyCallback`, `glfwSetMouseButtonCallback` and `glfwSetCursorPosCallback`
- `glfwSetWindowUserPoint` and `glfwGetWindowUserPointer`
- `glfwSetInputMode` with `GLFW_CURSOR`

The user pointer is an arbitrary pointer (e.g., to one of your own structures) that you can associate with a GLFW window. You can later retrieve the pointer from the GLFW window. Via this mechanism you can avoid having to rely on global variables to communicate the results from the callbacks back to your main program.

1.4 Mipmapping

4 marks

Extend the texturing to use use tri-linear (mipmapped) filtering.

You will need to generate the mipmap levels from the provided texture images.

For full marks, you should generate the mipmap levels using the Vulkan `vkCmdBlitImage` command during the loading phase of your application. Note that `vkCmdBlitImage` requires the source image subresource to be in the `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` layout and the destination image subresource to be in the `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`. This means that you need to transition the layout of individual mipmap levels. Use the `VkImageSubresourceRange` of the `VkImageMemoryBarrier` to do so.

Creating the mipmap levels on the CPU with your own code will result in fewer marks. Using third-party code or computing the mipmap levels offline ahead of time will result in the fewest marks. (If you do create the images ahead of time, include the created images in your submission.)

Do *not* resize the base level of the provided textures. Not all textures have a power-of-two texture size. Your solution/code will need to handle this. (Vulkan has no requirement to keep texture sizes or mipmap levels a power of two.)

1.5 Anisotropic filtering

3 marks

Extend the texturing to use anisotropic filtering. You will need to enable anisotropic filtering first when you create the logical Vulkan device, and later when creating the `VkSampler`.

In your submission, include two screen shots that illustrate the differences resulting from enabling anisotropic filtering.

2 Submission & Marking

In order to receive marks for the coursework, follow the instructions listed herein carefully. Failure to do so may result in zero marks.

Your coursework will be marked once you have

1. Submitted project files as detailed below on Minerva. (Do *not* send your solutions by email!)
2. Successfully completed a one-on-one demo session for the coursework with one of the instructors. Details are below - in particular, during this demo session, you must be able to demonstrate your understanding of your code. For example, the instructor may ask you about (parts of) your submission and you must be able to identify and explain the relevant code.
3. If deemed necessary, participated in an extended interview with the instructor(s) where you explain your submission in detail.

You do *not* have to submit your code on Minerva ahead of the demo session.

Project Submission Your submission will consist of a single `.zip`, `.tar.gz` or `.tar.bz2` file. Other archive formats are not accepted. The submission must contain your solution, i.e.,

- Buildable source code (i.e., your solutions and any third party dependencies). Your code must at the minimum be buildable and runnable on the reference machines in the Visualization Teaching Lab (with the exception that you may use Vulkan 1.3 core features, which are not available on those machines).
- A README as a structured `.pdf` or as properly formatted Markdown (`.md`). The README should identify what code you have written (i.e., is part of your solution) and what code (if any) is from third parties. The README must list which tasks (Section 1) that you have attempted. If a task requires you to specify additional information, you include this in the README.
- A list of third party components that you have used. Each item must have a short description, a link to its source and a reason for using this third party code. (You do not have to list reasons for the third party code handed out with the coursework, and may indeed just keep the provided `third_party.md` file in your submission.)
- The `premake5.lua` project definitions with which the project can be built.
- Necessary assets / data files.

Your submission *must not* include any unnecessary files, such as temporary files, (e.g., build artefacts or other "garbage" generated by your OS, IDE or similar), or e.g. files resulting from source control programs. (The submission may optionally contain Makefiles or Visual Studio project files, however, the program must be buildable using the included `permake5.lua` file only.)

If you use non-standard data formats for assets/data, it must be possible to convert standard data formats to your formats. (This means, in particular, that you must not use proprietary data formats.)

Demo Session The demo session will take either place in person or online during the standard lab hours.

For in-person demos, bring a physical copy of the *Demo Receipt* page (Appendix B), pre-filled with your information. The instructor will sign both sides, and keep the second half (the first half is for you).

Demo sessions will take place on a FIFO (first-in, first-out) basis in the scheduled hours. You might not be able to demo your solution if the session's time runs out and will have to return in the next session. *Do not wait for the last possible opportunity to demo your solution, as there might not be a next session in that case.*

A Rolling your own

These instructions apply if you do not wish to build on the provided code. If you build on the provided code, you can ignore this section.

When rolling your own code, there are a few additional requirements. First, you *must* inform the module's instructors ahead of time and get an OK. When getting in touch, explain why you wish to use your own code.

Additionally, when rolling your own, you will be responsible for being able to prove that you have written the code. The exact manner in which you would prove this is left up to you. If you are unsure about whether something counts as proof, you may ask the module's instructors.

It is your responsibility to ensure that your software builds and runs when tested by the module's instructors. The easiest way to do so is to ensure that it builds and runs on the machines in the Visualization Teaching Lab (2.16 in the Bragg building).

If you wish to use Vulkan 1.3 features, you will not be able to test your software on these machines. The instructors will have access to Vulkan 1.3 enabled-drivers. But you will not be able to test your work easily ahead of time.

You should strongly prefer to use the `premake` build system used with the exercises. If you do not do this, you must present a solid technical reason why the `premake` system is insufficient for your needs.

You must document any third party code that you use (link to source, license and short description). Minimize the amount of files that you distribute. For example, strip out unnecessary elements such as documentation, test cases or foreign language bindings. All necessary files must be included in your submission (i.e., your submission must be buildable on a machine that is *not* connected to the internet).

You are further required to describe the project layout in the README document in your submission.

You must still use the included assets, i.e., you will minimally require a functioning OBJ loader and reader for JPG images.

The instructors appreciate it if you keep your solution cross platform and/or runnable on Linux.

Your submission must be a fitting solution to the coursework. Under no circumstances should you submit a fully-featured engine (or similar) that just happens to also load and render the provided assets. Submissions that include code far outside of the scope of the tasks may receive deductions. Submissions that are extremely far outside of the scope of tasks in CW1 may not be marked at all.

B Demo Receipt

Please bring this page on an A4 paper if you are demo:ing your coursework in-person. Fill in the relevant fields (date, personal information) in both halves below. If the demo session is successful, an instructor will sign both halves. The top half is for your record. The instructor will take the bottom half and use it to record that you have successfully demo:ed your CW.

Please write legibly. :-)

Coursework 1 (Student copy)

Date . . . . ____________________

Name . . . . ____________________

UoL Username . . . . ____________________

Student ID . . . . ____________________

Instructor name . . . . ____________________

Instructor signature:

Coursework 1 (Instructor copy)

Date . . . . ____________________

Name . . . . ____________________

UoL Username . . . . ____________________

Student ID . . . . ____________________

Instructor name . . . . ____________________

Instructor signature: