School of Computing: assessment brief

Module title	Computer Graphics
Module code	COMP3811
Assignment title	This coursework covers the projective 3D graphics pipeline, which is fundamental to real-time rendering applications.
Assignment type and description	Programming assignment
Rationale	The coursework tests the student's ability create an interactive 3D rendering application, with custom objects and interactions.
Page limit and guidance	Report: 10 A4 pages with 2cm or larger margins, 10pt font size (including figures). You are allowed to use a double-column layout. Code: no limit. **Please read the submission instructions carefully!**
Weighting	50%
Submission deadline	2023-12-14 10:00
Submission method	Gradescope: code and report
Feedback provision	Written notes
Learning outcomes assessed	Understand, describe and utilize the projective 3D graphics pipeline and its modern implementation to create 2D images from 3D models; Understand, describe, apply and evaluate fundamental algorithms and methods from the field of computer graphics, commonly used for projective graphics applications such as 3D games.
Module lead	Markus Billeter

1. **Assignment guidance**

   In this second coursework, you are tasked with developing an OpenGL-based renderer in groups of three students. You will develop an application that combines a terrain based on real-world elevation data with pre-made models loaded from files and models programmatically created by you.

   Before starting your work, please study the coursework document in its entirety. Pay special attention to the requirements and submission information. Plan your work. It is likely better to focus on a subset of tasks and commit to these fully than to attempt everything in a superficial way.

2. **Assessment tasks**

   Please see detailed instructions in the document following the standardized assessment brief (pages i-v). The work is split into several tasks, accounting in total for 50% of the module grade.

3. **General guidance and study support**

   Please refer to materials handed out during the module, including the tutorial-style exercises for the 3D graphics pipeline.

   Support is primarily provided during scheduled lab hours. Support for general issues may be provided through the module's "Teams" channel. Do not expect answers outside of scheduled hours. Do not post specific issues relating to your code in the public Teams channels. Do not crosspost across multiple channels.

4. **Assessment criteria and marking process**

   Submissions take place through Gradescope. Valid submissions will be marked primarily based on the report and secondarily based on the submitted code. See following sections for details on submission requirements and on requirements on the report. Marks and feedback will be provided through Minerva (and not through Gradescope - Gradescope is only used for submissions!).

5. **Submission requirements**

   Your coursework will be graded once you have

   (a) Submitted required files (code and report) through Gradescope.

   (b) If deemed necessary, participated in an extended interview with the instructor(s) where you explain your submission in detail.

   Your submission will consist of source code and a report ($\lesssim 10$ pages). *The report is the basis for assessment. The source code is supporting evidence for assertions made in the report.*

   Submissions are made through Gradescope (do *not* send your solutions by email!). You can use any of Gradescope's mechanisms for uploading the complete solution

and report. In particular, Gradescope accepts `.zip` archives (you should see the contents of them when uploading to Gradescope). Do not use other archive formats (Gradescope must be able to unpack them!). Gradescope will run preliminary checks on your submission and indicate whether it is considered a valid submission.

The source code must compile and run as submitted on the standard SoC machines found in the UG teaching lab (2.05 in Bragg). Your code must compile cleanly, i.e., it should not produce any warnings. If there are singular warnings that you cannot resolve or believe are in error, you must list these in your report and provide an explanation of what the warning means and why it is acceptable in your case. This is not applicable for easily fixed problems and other bulk warnings (for example, type conversions) – you are always expected to correct the underlying issues for such. *Do not change the warning level defined in the handed-out code. Disabling individual warnings through various means will still require documenting them in the report.*

In general, you must not use custom extensions or vendor-specific features in your application (foe example, NVIDIA-specific OpenGL exensions). If you wish to explore a specific advanced feature that requires such, please conctact the module leader ahead of time and get their explicit permission. Contact the module leader by email for this, explaining what feature you require, what you aim to do (and why that specific feature is required for that).

Your submission must not include any "extra" files that are not required to build or run your submission (aside from the report). In particular, you must *not* include build artifacts (e.g. final binaries, `.o` files, ...), temporary files generated by your IDE or other tools (e.g. `.vs` directory and contents) or files used by version control (e.g. `.git` directory and related files). Note that some of these files may be hidden by default, but they are almost always visible when inspecting the archive with various tools. Do not submit unused code (e.g. created for testing). Submitting unnecessary files may result in a deduction of marks.

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****. Members of your group should be the only users with access to that repository.*

6. **Presentation and referencing**

Your report must be a single PDF file called `report.pdf`. In the report, you must list all tasks that you have attempted. *Include screenshots for each task unless otherwise noted in the task description!* You may refer to your code in the descriptions, but descriptions that just say "see source code" are not sufficient. Do **not** reproduce bulk code in your report. If you wish to highlight a particularly clever method, a short snippet of code is acceptable. Never show screenshots/images of code - if you wish to include code, make sure it is rendered as text in the PDF using appropriate formatting and layout. Screenshots must be of good quality (keep the resolution at 1280×720

or higher, but scale them down in the PDF). Don't compress the screenshots overly much (e.g., visible compression artifacts).

Apply good report writing practices. Structure your report appropriately. Use whole English sentences. Use appropriate grammar, punctuation and spelling. Provide figure captions to figures/screenshots, explaining what the figure/screenshot is showing and what the reader should pay attention to. Refer to figures from your main text. Cite external references appropriately.

Furthermore, the UoL standard practices apply:

The quality of written English will be assessed in this work. As a minimum, you must ensure:

- Paragraphs are used
- There are links between and within paragraphs although these may be ineffective at times
- There are (at least) attempts at referencing
- Word choice and grammar do not seriously undermine the meaning and comprehensibility of the argument
- Word choice and grammar are generally appropriate to an academic text

These are pass/ fail criteria. So irrespective of marks awarded elsewhere, if you do not meet these criteria you will fail overall.

7. **Academic misconduct and plagiarism**

   You are encouraged to research solutions and use third-party resources. If you find such, you must provide a reference to them in your report (include information about the source and original author(s)). Never "copy-paste" code from elsewhere – all code must be written yourself. If the solution is based on third party code, make sure to indicate this in comments surrounding your implementation in your code, in addition to including a reference in your report.

   If you wish to use a custom third party library, you must clear this with the instructors for COMP3811 in writing well ahead of your submission. You must provide a clear reason for using this library in your request. Third party libraries may not be used to solve substantial portions of any of the tasks and cannot be used to replace elements provided with the base code.

   Furthermore, the UoL standard practices apply:

   Academic integrity means engaging in good academic practice. This involves essential academic skills, such as keeping track of where you find ideas and information and referencing these accurately in your work.

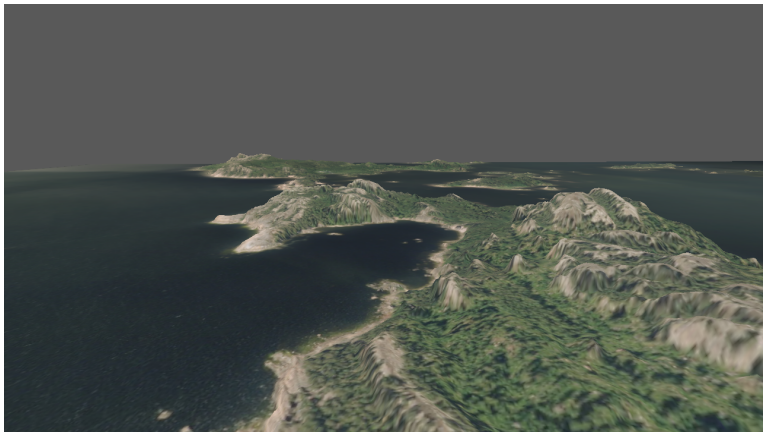
By submitting this assignment you are confirming that the work is a true expression of your own work and ideas and that you have given credit to others where their work has contributed to yours.

8. **Assessment/marking criteria grid**

   (See separate document.)

COMP3811 Coursework 2

Contents



In this coursework, you will create a renderer using the (modern) OpenGL API. The coursework is to be completed in **groups of three**. You must build on the code provided with this coursework. In particular, the project must be buildable through the standard premake steps (see exercises). Carefully study the submission requirements for details.

*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. The members of the group should be the only users with access to that repository.*

1 Tasks

The total achievable score in Coursework 2 is **50 marks**. The coursework is split into the tasks listed below. Please study all tasks and plan your work before you start coding. It is likely better to focus on a subset of tasks and commit to these fully than to attempt everything in a superficial way.

Background: The privately held space company, AvaruusY, is preparing for lift-off. As part of their preparations, they are looking for an interactive 3D experience that lets the public explore their launch. You have been contracted for this job! The launch is planned to take place from a sea-based launch platform in southern Finland. AvaruusY engineers have spent the last few days furiously compiling the data that they have about the launch site and have produced a few 3D models for your use.

Important: In the end of the report (appendix), you must include a table that lists each group member's individual contributions to each of the completed tasks. Document who wrote what code. Make sure that all group members contribute substantially to the code. Prefer working together on tasks rather than distributing them to individuals. When multiple members worked on a feature (e.g., pair programming), include rough proportions.

1.1 Matrix/vector functions

4 marks

Your first task is to implement the necessary matrix and vector functions in `vmlib`. Minimally, this includes the following functions in `vmlib/mat44.hpp`:

- `Mat44f operator*( Mat44f const&, Mat44f const& )`
- `Vec4f operator*( Mat44f const&, Vec4f const& )`
- `Mat44f make_rotation_{x,y,z}( float )` (three functions)
- `Mat44f make_translation( Vec3f )`
- `Mat44f make_perspective_projection( float, float, float, float )`

These functions are quite fundamental and needed for the rest of this coursework. It is therefore important to be able to rely on them. To help you check for the correctness, the handed-out code includes the `vmlib-test` subproject. It uses the Catch2 testing library. A sample test case for the standard projection matrix is included as an example.

You should add tests for each of the above functions. Comparing against known-good values is sufficient in this case (but make sure your known-good values are indeed correct!). Deriving the known-good values by hand or using a (well-known) third-party library are both acceptable. Make sure your tests are meaningful.

Note: you are required to use the `vmlib` types in your solution – i.e., you may not use a third party math library!

In your report: Briefly outline the tests that you have added. Explain why those tests are meaningful. (No screenshots are required.)

1.2 3D renderer basics

8 marks

Next, set up a 3D rendering application using modern shader-based OpenGL in the `main` subproject. You may start with the skeleton provided in `main/main.cpp`. Refer to the OpenGL exercises from the module as necessary.

The base code includes a Wavefront OBJ file, `assets/parlahti.obj` and `assets/parlahti.mtl`. The mesh shows a small part of Finland, near the Parlahti region with the islands Heinäsaari, Sommarön and Långön (*saari* is island in Finnish; *ö* is island in Swedish). The region is 6km by 6km and derived from a digital elevation model with 2m resolution. The elevation is somewhat exaggerated in the model, to make the terrain more visible. The original elevation data was obtained from *Maanmittauslaitos*, the national land survey of Finland. You can download other parts of Finland from their homepage. The datasets are subject to the Creative Commons Attribution 4.0 license.

You should set up a program that can load this Wavefront OBJ file and display it.

Use a perspective projection for rendering. You must use the `make_perspective_projection` function from `vmlib` to create the projection matrix. Make sure the window is resizable.

Implement a camera with which you can explore the scene using the mouse and keyboard. Aim for a first-person style 3D camera (WSAD+EQ to control position, mouse to look around, shift to speed up, and ctrl to slow down). Camera controls should be frame-rate independent. Implement this using the callback-style input from GLFW – do not use polling for mouse and keyboard input (so, no `glfwGetKey` and similar).

Use the simplified directional light model from Exercise G.5, i.e., with only a ambient and diffuse component. For now, use the light direction $(0, 1, -1)$ (but remember to normalize it!).

Refer to Figure 1 for some sample screenshots of what this may look like.

In your report: Report the values for `GL_RENDERER`, `GL_VENDOR` and `GL_VERSION` for your test computer(s). Include several (3+) screenshots of the scene. They must be significantly different (=different views) from the ones shown in Figure 1.

1.3 Texturing

4 marks

In addition to the elevation data, *Maanmittauslaitos* also provides orthophotos. These are photos taken with a top down view (for example, from an airplane). We can use this orthophoto as a texture for the terrain mesh.

The Wavefront OBJ includes all the necessary data. Update your renderer to draw the mesh with a texture. Combine the texture with the simple lighting from Section 1.2.

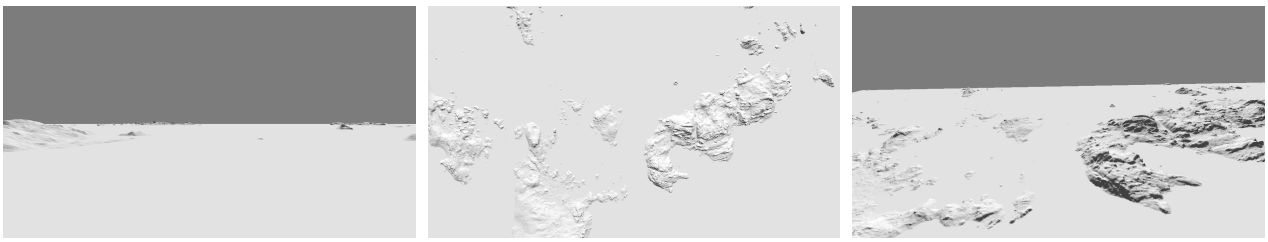


Figure 1: *Screenshots showing the Parlahti mesh, with a simple "n dot l" lighting model. The simple lighting model helps us see the geometry clearly.*

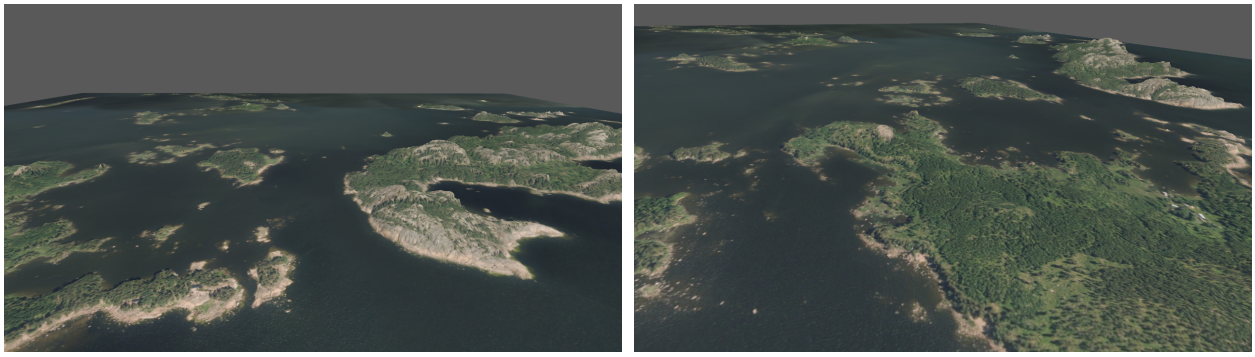


Figure 2: *Screenshots showing the Parlahti model with the orthophoto-based texture. Note that it does not include any lighting. (You should however combine the texture with lighting in your implementation!)*

See Figure 2 for a textured rendering (no lighting).

In your report: Include several (3+) screenshots with texturing. They must be significantly different (=different views) from the ones shown in Figure 2.

1.4 Simple Instancing

4 marks

"Why build one when you can have two at twice the price?", Contact (1997)

The exercise includes a second Wavefront OBJ file, `assets/launchpad.obj` and `assets/launchpad.mtl` (Figure 3). This models the launchpad that AvaruusY is planning to use for its launches.

Load this file (once). Find **two** spots for the launch pad. They must be in the sea and in contact with the water (but not fully submerged). They cannot be at the origin. They should not be immediately next to each other. Render the object twice (without making a copy of the data), once at each location. The model does not have a texture, but uses per-material colors. You will need to handle this somehow, for example with two different shader programs.

In your report: Document the coordinates at which you placed the launch pads. Include screenshots of the placement.

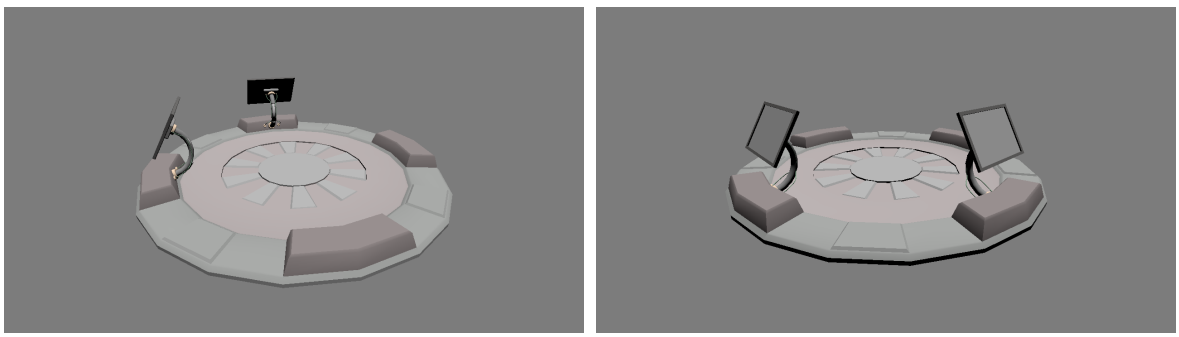


Figure 3: *Launchpad model (`launchpad.obj`).*

1.5 Custom model

4 marks

Create a 3D model of a space vehicle programmatically. Combine basic shapes (boxes, cylinders, ...) into a complex object. You must use at least seven (7) basic shapes. These must be connected to each other (e.g., they can not float freely) and all of the parts must be visible. You must use at least three (3) different basic shapes. Simple transformations of a shape, such as a non-uniform scaling, do not constitute a different shape (for example, transforming a cube into a box still only counts as one shape). You should however use all transformations (translation, rotation, scaling) when placing objects. At least one object must be placed relative to another (e.g., you place one thing on top of another).

Make sure you generate the appropriate normals for the object, such that it is lit correctly. You do not have to texture the object. You are free to use different colors for different parts if you wish.

The space vehicle must be placed on one of the launch pads (Section 1.4).

Note: You have some freedom in designing your "space vehicle". It should be recognizable as a vehicle, but it does not have to be a rocket. An alien UFO would work as well. Or a submarine ... after all, that's a bit like a spaceship, except that it goes the other way[1].

In your report: Document which of the launch pads you placed your space vehicle on. Include screenshots (2+) of your vehicle.

1.6 Local light sources

3 marks

Implement the full Blinn-Phong shading model for point lights, including the standard $1/r^2$ distance attenuation. Add three point lights, each with a different color, to your space vehicle (Section 1.5). You may place them slightly away from the geometry to avoid problems with the division.

In your report: Include a screenshot of lit space vehicle and launch pad.

1.7 Animation

3 marks

Implement an animation where your space vehicle flies away. The animation should start when the user presses the `F` key. Pressing `R` should reset the space vehicle to its original position and disable the animation.

For full marks, you should consider a curved path for the vehicle, where it slowly accelerates from a standstill. The vehicle should rotate such that it always faces into the direction of movement. The lights from Section 1.6 should follow the space vehicle.

In your report: Include a screenshot of the vehicle mid-flight.

1.8 Tracking cameras

3 marks

Implement a feature where the user can switch to cameras that track the space vehicle in its flight (Section 1.7). Implement two cameras modes:

- A camera that looks at the space vehicle from a fixed distance and follows it in its flight.
- A camera that is fixed on the ground and always looks towards the space vehicle, even as it flies away.

The user should be able to cycle through the different camera modes with the `C` key. So, pressing the key should first switch from the freely controllable default camera to the fixed-distance camera, next to the ground camera and then back.

In your report: Include a screenshot of each camera mode, with the vehicle in mid-flight.

1.9 Split screen

4 marks

Implement a split screen rendering mode that shows two different views at once. Let the user toggle split screen mode by pressing the `V` key. Pressing `C` should select the camera mode for one of the views (Section 1.8). The camera mode for the other one should cycle by pressing `Shift-C`. You can pick how you split the screen, but a simple horizontal 50%-50% split is fine.

[1] The AvaruusY engineers might object to a car, though, since a different company already sent one of those to space.

There are multiple possible strategies for implementing a split screen mode. You are free to pick one yourself. However, it should not be overly wasteful (so don't render parts that are never shown to the user). The split screen mode should adapt to the window when resized.

In your report: Describe how you have implemented the split screen feature. Include screenshots that show the split screen mode in action.

1.10 Particles

4 marks

Implement a particle system that simulates exhaust from your space vehicle as it flies away (Section 1.7).

For full marks, consider the following features:

- Render particles as textured point sprites or as textured camera-facing quads.
- Make sure the particles render correctly from all views. Designing the system around additive blending will make this easier.
- Particles should be emitted in whatever small area you consider the vehicle's engine, and should appear to move away from the vehicle.
- Particles should have fixed lifetime after which they disappear.

Take particular care with the implementation. For example, avoid dynamic allocations at runtime during the simulation. A CPU-based implementation is perfectly adequate.

In your report: Describe how you have implemented the particle system. Document any assumptions that you have made and mention limitations. Discuss the efficiency/performance of the system. Include a screenshot of the effect.

1.11 Simple 2D UI elements

4 marks

Implement the following UI elements:

- In the top left corner, display the current altitude of the space vehicle in text.
- In the bottom center, implement two buttons. One that launches the space vehicle when clicked and one that resets the vehicle to its initial position (see Section 1.7).

Make sure the buttons stay in the right place when the window is resized.

Draw the buttons such that they their label is centered in a rectangle with a solid outline and a semi-transparent fill. The buttons should have three distinct visual states: normal, mouse-over and pressed (mouse button held down). The top-left status text should remain readable at all times.

You should implement this yourself, e.g., without a third party UI library. You may however use Fontstash for rendering text (use the included `assets/DroidSansMonoDotted.ttf` font). You can also opt for rendering text using a font atlas texture if you prefer.

In your report: Describe your implementation. What are the steps to adding another UI element? Include screenshots of the UI, including the different button states.

1.12 Measuring performance

5 marks

Measure the performance of your renderer. This task primarily focuses on the GPU rendering performance (recall that the GPU will run asynchronously from the CPU). You must therefore use `GL_TIMESTAMP` queries with the `glQueryCounter()` function. Study the documentation for the above function. Refer to OpenGL wiki on query objects for additional information.

Use the queries to measure rendering performance. Each frame, you should measure:

- the full rendering time (but exclude e.g., `glfwSwapBuffers()` and other non-rendering code).
- the time needed to render the parts from Sections 1.2, 1.4 and 1.5 *individually*, so as three different times.

If you implemented Section 1.9, you can measure this only for one of the views or for both individually. For full marks, you should implement a strategy that avoids stalling/waiting when retrieving results from the asynchronous timestamp queries. No marks will be awarded if you use third party applications to measure the performance.

Additionally measure frame-to-frame times and the time it takes to submit the rendering commands with OpenGL using a CPU timer such as `std::chrono::high_resolution_clock`.

Perform the measurements over multiple frames. Make sure you run in release mode and that you have disabled any OpenGL debugging and removed any potentially blocking calls (e.g., `glGetError()`). Ideally, try to run this on one of the newer machines in 2.05, e.g., with the NVIDIA RTX GPUs.

In your report: Describe your implementation. Document your results (table or plot) and discuss these. Compare the different timings. Are they what you would expect? Are the timings reproducible in different runs? Do they vary as you move around? Explain your observations. (No screenshots required.)

2 Implementation Requirements

For full marks, you are expected to following the requirements/specification detailed in this document. Further, you are expected apply to good (graphics) engineering practices, including:

- Manage resources! In debug mode, you should clean up all resources and allocations. In release mode, you may choose to "leak" some resources on exit as an optimization (which the OS will free up once the process terminates), but in order to aid debugging, you must not do so in debug builds. Furthermore, you should never leak run-time allocations that cause resource usage to continually increase.
- Do not unnecessarily duplicate data. Typically, any large resource should be allocated at most once and never be duplicated. If you duplicate resources, you should make this an informed choice and give a reason for doing so (e.g., double-buffering due to concurrent accesses, or in order to enable batching for more efficient draw calls).
- Do not do unnecessary work per frame. Avoid allocating and deallocating data each frame; allocate upfront and re-use resources instead. Don't do file IO per frame. Don't repeatedly query data that doesn't change (for example, don't `glGetUniformLocation` each frame!).
- Do not do unnecessary work in the pipeline: Don't do computations in the fragment shader if they could be done in the vertex shader. Don't do computations in the vertex shader if they could be done once on the CPU. Don't repeat computations each frame if they can (reasonably) be done once on startup. Etc. (This is sometimes a trade-off between memory and compute. If in doubt, motivate your choice.)
- Use appropriate data structures. For example, by default you should prefer an `std::vector` over a linked list. Don't use a `shared_ptr` when an `unique_ptr` would do. ...
- Don't reinvent the wheel unnecessarily. For example, do use the C++ standard library's containers, rather than implementing your own. (The exercise requires you to implement your own vector/matrix functions. This can be considered a special exception to this guideline.)

Wrapping up

Please double-check the requirements in and ensure that your submission conforms to these. In particular, pay attention to file types (archive format and report format) and ensure that you have not included any unnecessary files in the submission.

Make sure you have included the table with individual contributions in your report (see Section 1).