Task 1 – Matrix/Vector functions

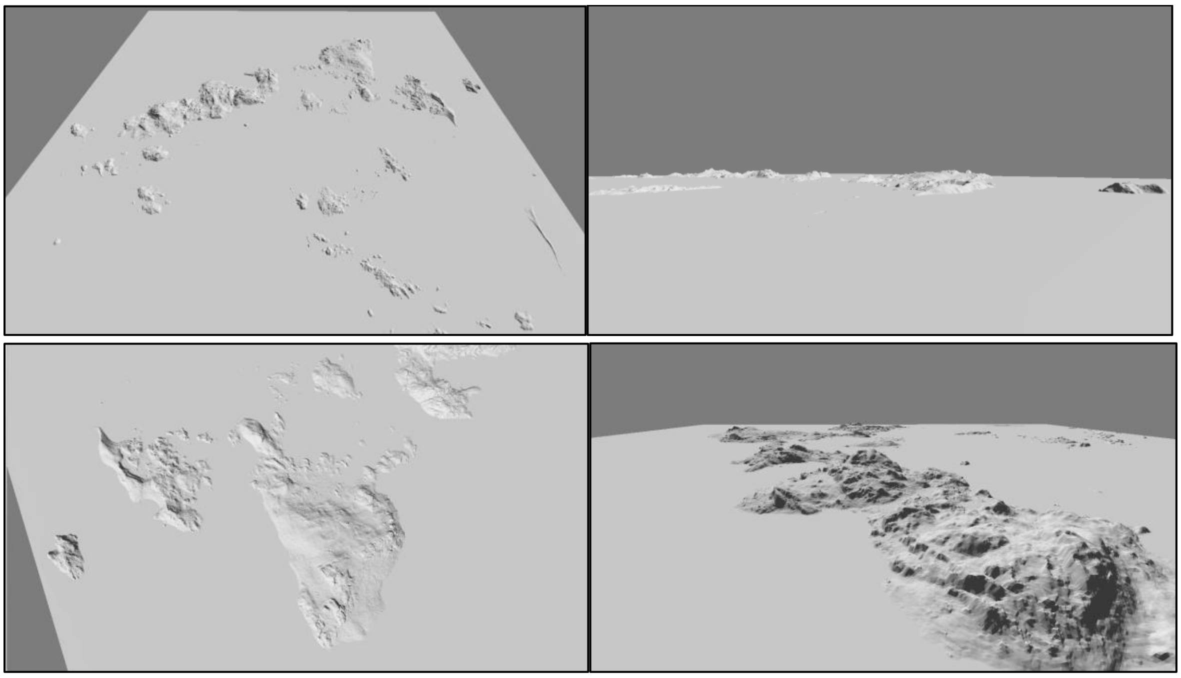
The functions we have implemented are the operator* functions, make_rotation_{x, y, z}, make_translation, and make_perspective_projection.

To test the multiplication overload functions, we created two test cases for matrix-matrix multiplication and matrix-vector multiplication. For each test case, we tested multiplication with an identity matrix and zero matrix to ensure that it is mathematically correct and tested for expected results that were calculated by hand to ensure correctness.

To test the rotation functions, we tested each quadrant of rotation for correctness. The cases for 0- and 360-degree rotations are also edge cases that return the identity matrix. We did these for each axis.

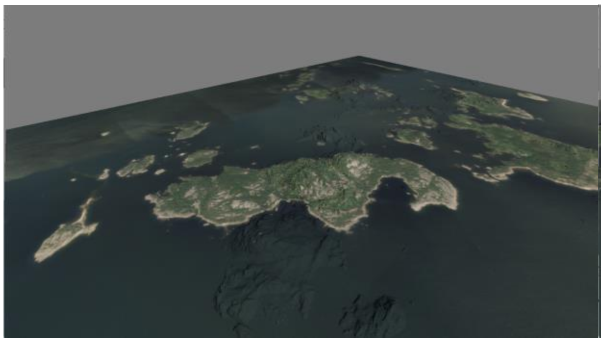
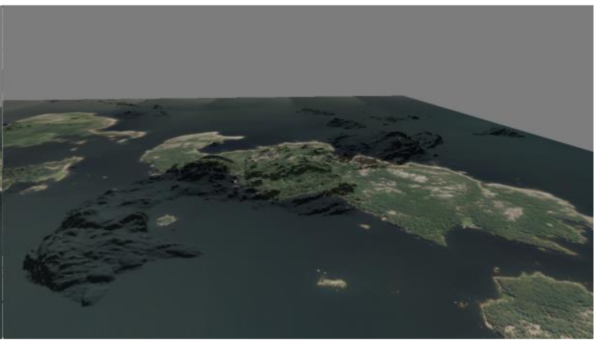
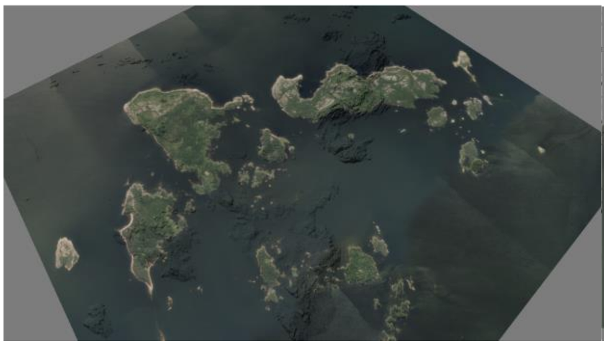
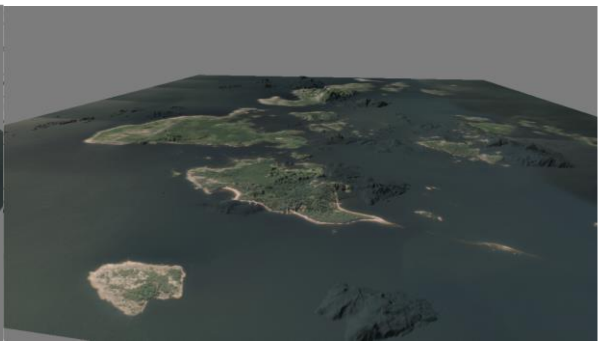
To test translation, we tested for a translation of 0 on all axis to produce an identity matrix. This verifies that the position remains unchanged when there is no translation. We also tested positive and negative values in translation, to make sure it handles both directions.

Task 2 – 3D renderer basics



	Iven	Aik
GL_RENDERER	AMD Radeon RX 7800 XT	Intel(R) HD Graphics 620
GL_VENDOR	ATI Technologies Inc.	Intel
GL_VERSION	4.3.0 Core Profile Context 25.11.1.51031	4.3.0 - Build 27.20.100.8682

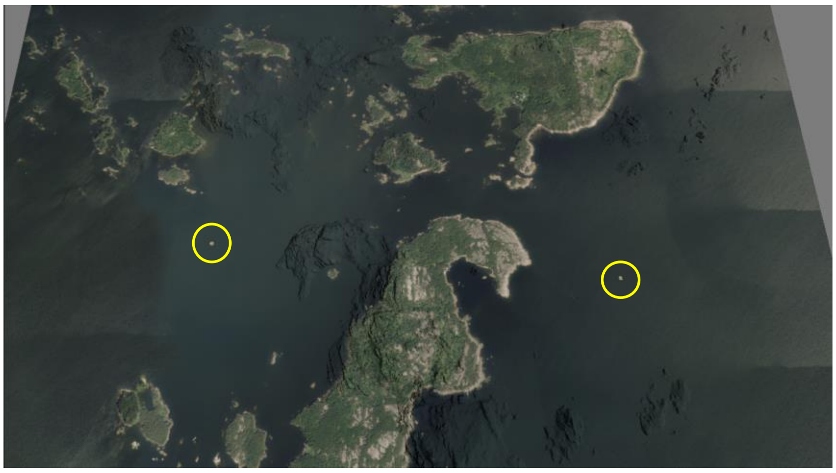
Task 3 – Texturing

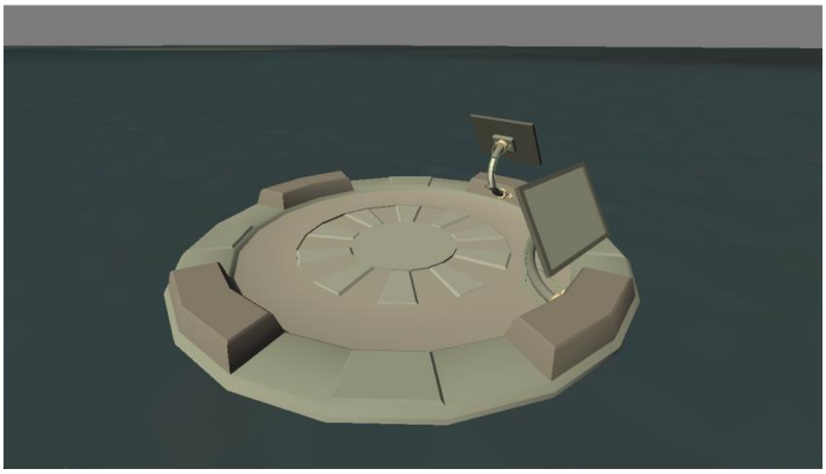





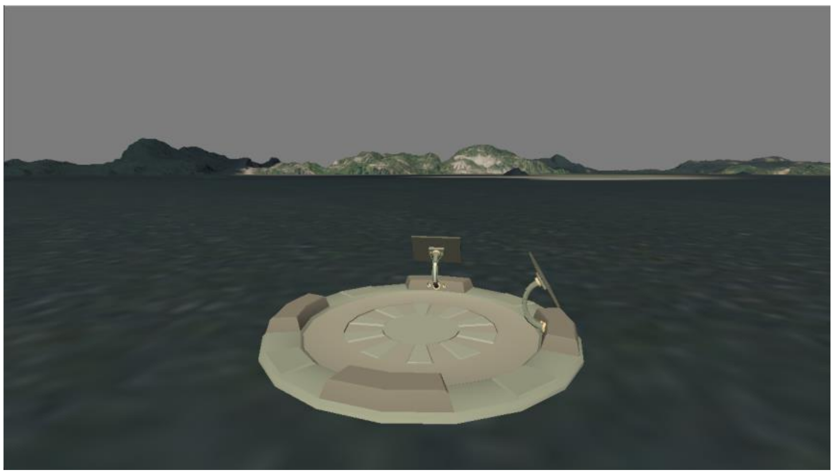
Task 4 – Simple Instancing

Landing pad coordinates:

- (10, -0.95, 45)
- (20, -0.95, -50)

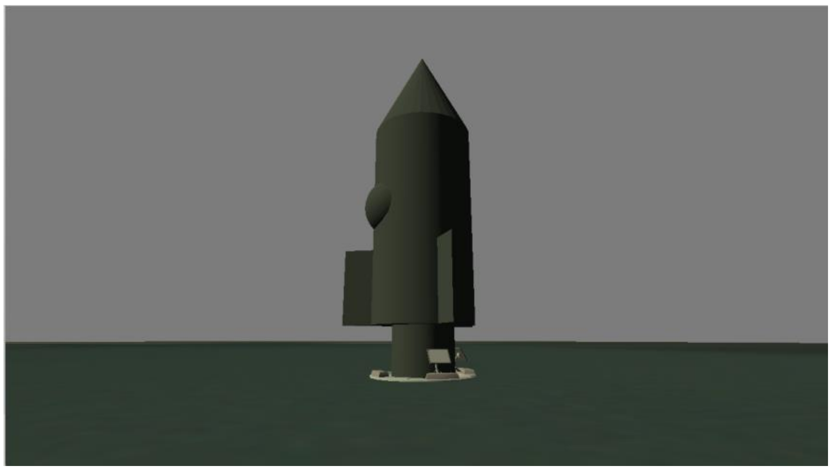


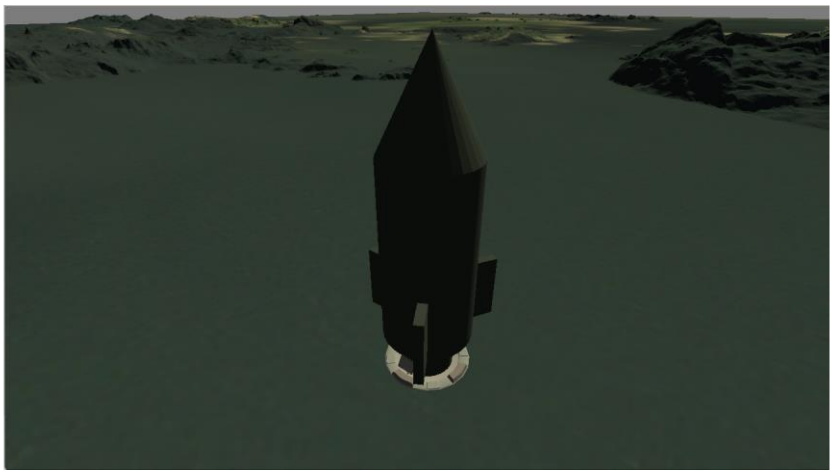


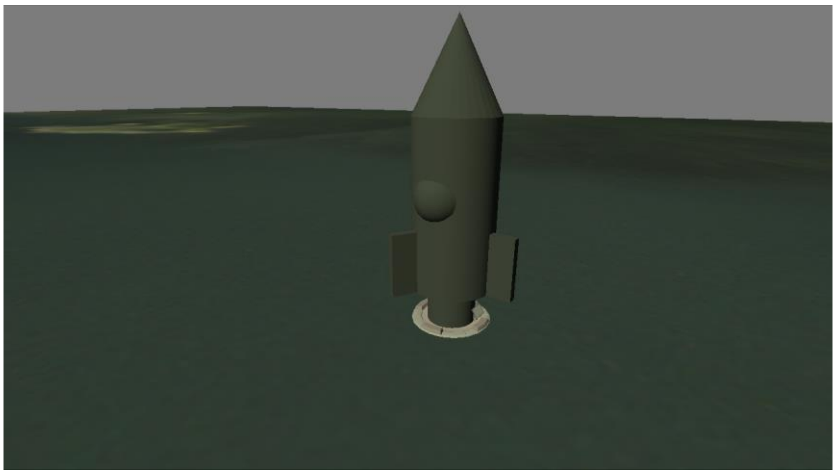


Task 5 – Custom model

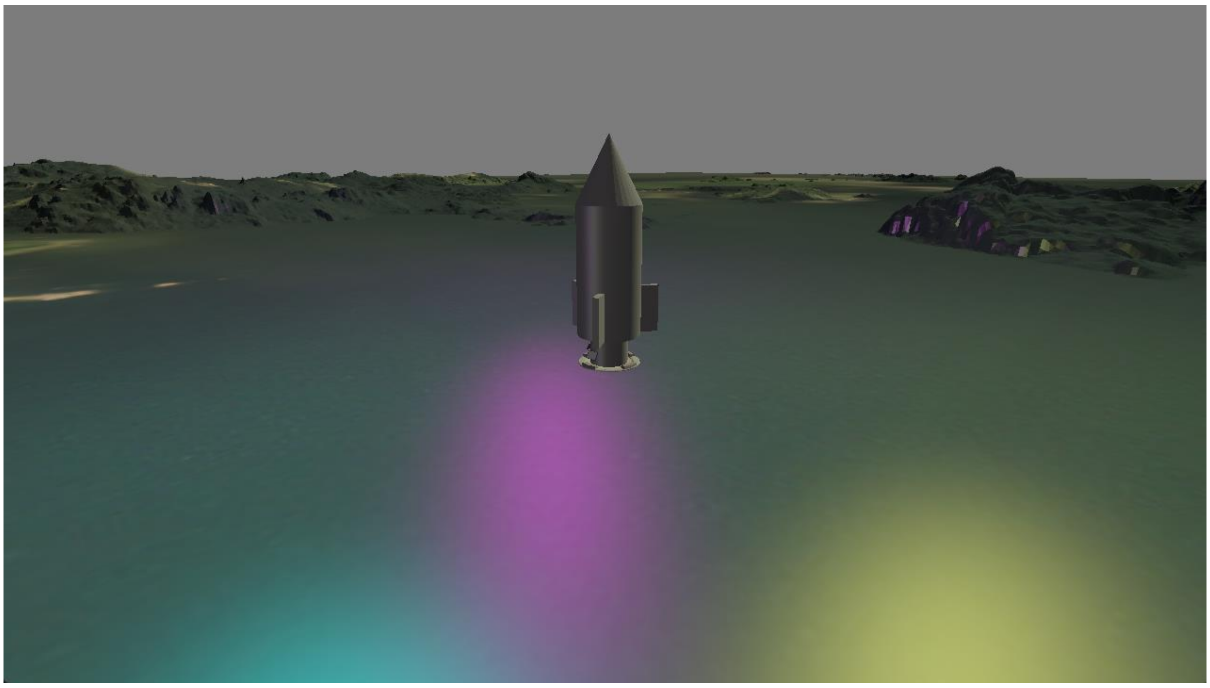
Placed space vehicle on landing pad with coords: (10, -0.95, 45).



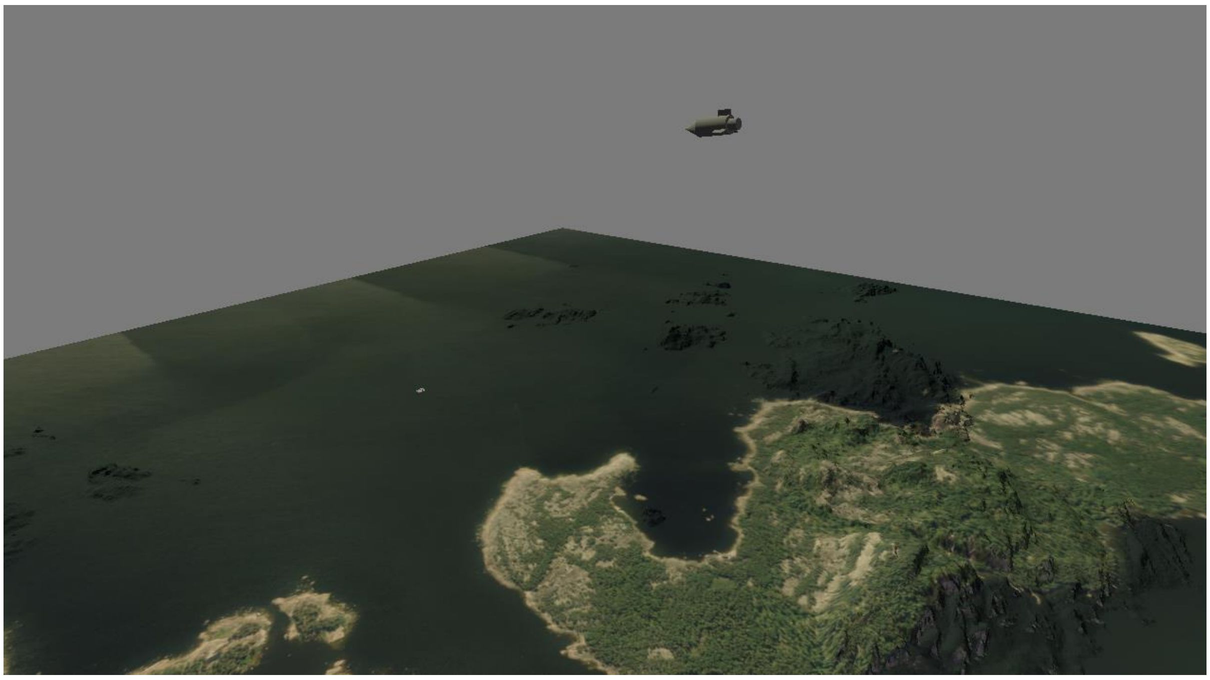




Task 6 – Local light sources



Task 7 – Animation



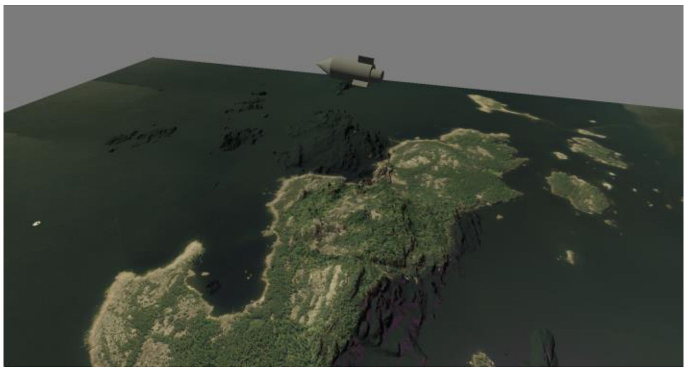
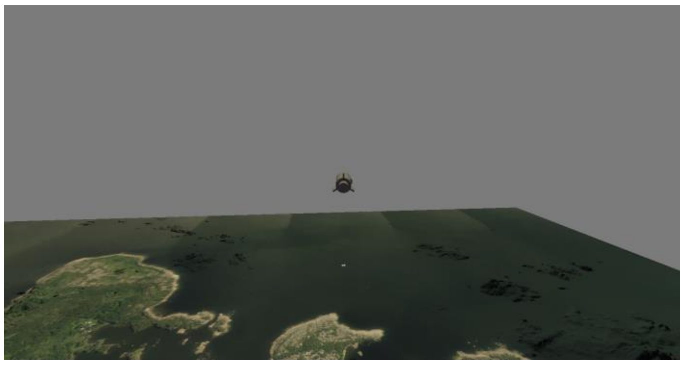
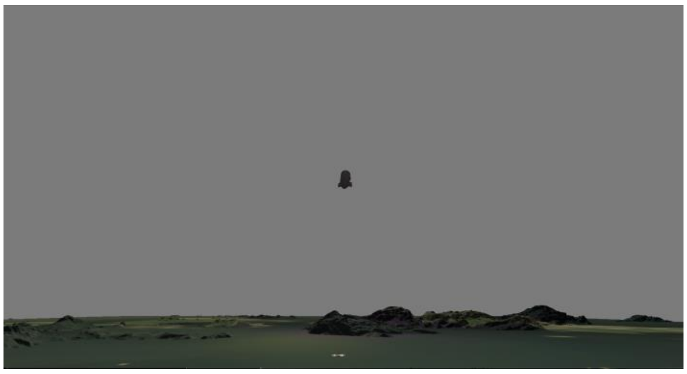
Task 8 – Tracking cameras

3 camera modes switched when C is pressed:

Free-roam – WASD callbacks

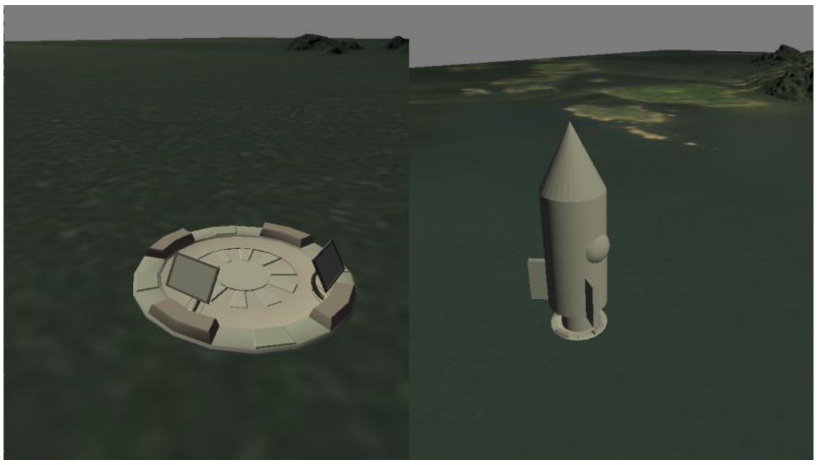
Chase – follow the rocket 20 units behind (disabled when the animation is inactive)

Fixed – Fixed spot on the ground which is offset from the launch pad

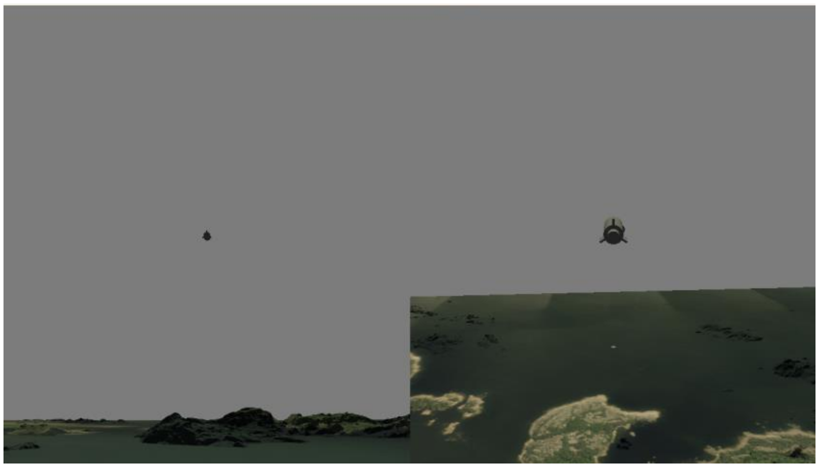
Task 9 – Split screen

The split screen feature was implemented by rendering the entire scene twice using a single function, to prevent code duplication. The Boolean variable ‘splitScreen’ toggles the feature on and off, whenever the user presses ‘v’. Each screen has its own camera position and mode, which is tracked by using the same type of structs as the single screen implementation. To display both screens, the window width is halved and used in the projection matrix and glViewPort to define the area for each screen's rendering. Both screens behave the same and use the same inputs from the user. This includes inputs for free cam movement, which only updates the screen-specific camera position if free cam is the current mode of that screen. The exception to the same inputs is the cycling of camera modes, which is done with ‘c’ and ‘shitf+c’ for the left and right screen respectively.



Left: Landing pad at (20, -0.95, -50)

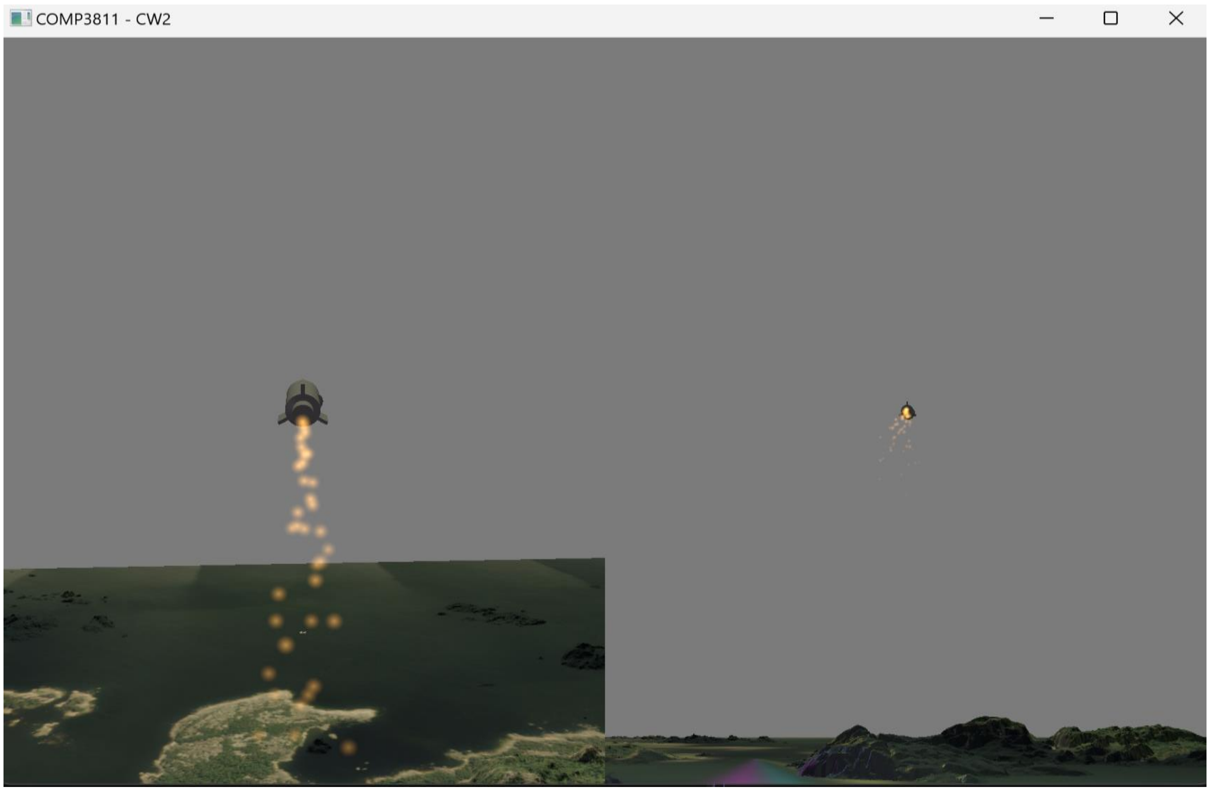
Right: Landing pad with spaceship at (10, -0.95, 45)



Left: Fixed camera during spaceship animation.

Right: Chase camera during spaceship animation.

Task 10 – Particles

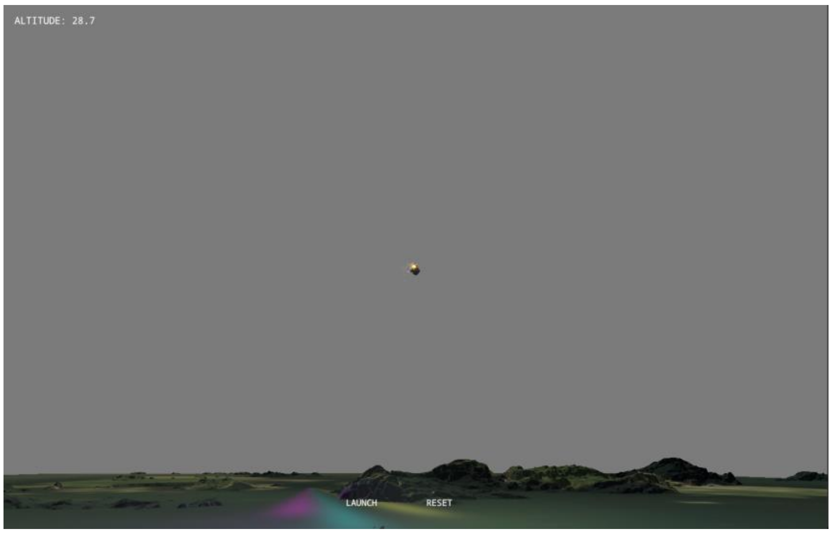


The particle system was implemented as a simple exhaust effect for the space vehicle. Particles are spawned from the bottom of the rocket when the launch animation is active and are given random velocities and lifetimes to simulate thrust dispersal. Each frame, particles are updated on the CPU by moving them based on velocity, applying gravity, and removing them once their lifetime expires.

Particles are rendered as camera-facing "fuzz balls" using additive blending to create a glowing, semi-transparent exhaust appearance suitable for game-like visuals. During rendering, depth testing is enabled so particles are correctly hidden behind solid objects, while depth writing is disabled to prevent particles from occluding each other, ensuring smooth blending.

The system assumes a limited number of particles (capped at 1024) and performs all updates on the CPU. This is efficient enough for the scope of the coursework, but it would not scale well for larger or more complex particle effects and would lead to reduced performance in more demanding scenarios.

Task 11 – Simple 2D UI elements



A minimal text based, hardcoded UI overlay was implemented using the provided Fontstash library. The UI is rendered in screen space and drawn after the 3D scene to ensure it remains visible regardless of camera position.

Fontstash converts text characters into atlas textures during runtime. Custom OpenGL callbacks were implemented to create, update, and render this atlas using a simple shader that normalizes pixel coordinates.

The UI displays an altitude HUD at the top-left of the screen and two interactive text buttons, LAUNCH and RESET, positioned near the bottom center. UI elements automatically adjust to window resizing by recalculating positions using the current framebuffer dimensions.

Mouse input is handled via GLFW callbacks, with click detection implemented using simple screen-space rectangular hitboxes. Clicking the text buttons triggers the same animation logic as the keyboard controls, ensuring consistent behaviour.

The UI does not include button rectangles, hover states, or visual feedback beyond text, as these were omitted due to time constraints and implementation complexity.

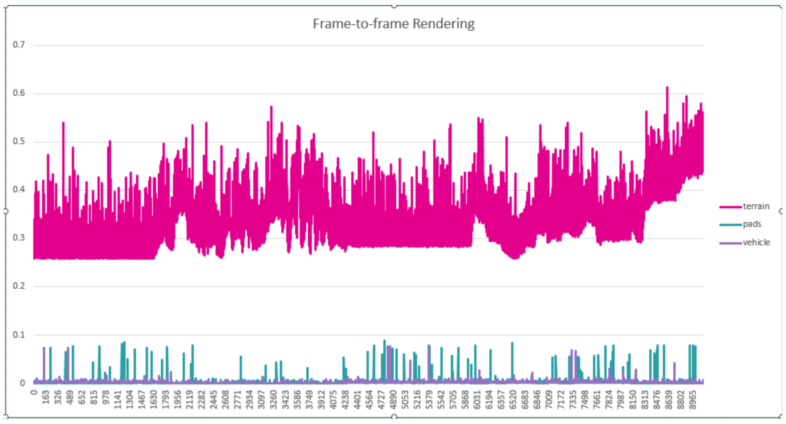
To add a new UI element, the element is first rendered in screen space using Fontstash inside the ui_draw function. Interaction logic is then added to the ui_mouse_button callback by defining a simple screen-space hitbox that mirrors the approach used for the existing LAUNCH and RESET buttons.

Task 12 – Measuring performance

To measure performance, a struct 'GPUTimers' was implemented. This struct contains a vector of OpenGL query IDs to execute timestamp queries at the required sections of measurement, and a vector of the struct 'Result'. As the name suggests, the 'Result' struct stores the result of the queries – namely the raw timestamps, computed differences between timestamps and the corresponding frame. Since GPU queries are asynchronous, timestamp reads occurred a fixed number of frames later, controlled by the 'ringSize' variable in the 'GPUTimers' struct. This ensured that the data wasn't affected by blocking calls that would stall the GPU.

Component	**Maximum (ms)**	**Minimum (ms)**	**Average (ms)**
Terrain	0.64264	0.22684	0.30846
Landing Pads	0.08300	0.00008	0.00391
Space Vehicle	0.11592	0.00008	0.00431
Full rendering	0.64632	0.23540	0.31688

From the table above, we can see that the terrain takes up most of the rendering time, which is expected since it has nearly 3 million vertices compared to the landing pads and space vehicle having roughly 19,000 and 3000 vertices respectively. The variation of each component is also expected to be normal, since frame-to-frame rendering can be inconsistent depending on the current scene (camera movement/view, lighting) and GPU scheduling.



In the graph above, the rendering time for the terrain started fluctuated two times. For the first fluctuation, the camera was in constant movement via mouse movement and keyboard inputs and in the second fluctuation, both the global lights and local lights were toggled on and off in quick succession. This is expected since camera movement changes which fragments are visible and lighting changes how the existing fragments will appear.

#ifdef Macro used is ENABLE_GPU_TIMERS. Uncomment the #define at the top of main.cpp to toggle the code for measurements.

Appendix

Task	**Subtask**	**Contributor**
1.1	Implementing operator* functions.	sc23mas
	Implementing make_rotation_{x, y, z} functions.	sc23mas
	Implementing make_translation function.	sc23mas
	Implementing make_perspective_projection function.	sc23mas
	Writing test code for above functions.	sc23imm
1.2	Rendering the terrain mesh.	sc23mas
	Camera with controls.	sc23imm
	Simple lighting and Blinn_Phong shading.	sc23imm
1.3	Adding texture to the terrain mesh.	sc23mas
1.4	Rendering double instance of landing pad.	sc23imm
1.5	Custom model of spaceship.	sc23mas
1.6	Local lights sources.	sc23imm
1.7	Spaceship animation.	sc23mas
1.8	Cameras tracking the spaceship animation.	sc23mas
1.9	Split screen feature.	sc23imm
1.10	Animation particles.	sc23mas
1.11	2D UI Elements (Partially complete)	sc23mas
1.12	Measuring rendering time of GPU.	sc23imm