Task 1: Matrix/Vector Functions

For this task, the operators for matrix-matrix and matrix-vector multiplication, rotation in the X, Y and Z planes, translation, and perspective projection matrices have been implemented. The tests for matrix-matrix multiplication run through some basic known properties, including identity matrix multiplication, the associative property, communicative property for diagonal matrices only, and multiplication by a zero-matrix. For matrix-vector multiplication, we test by: multiplying against the zero-matrix; identity matrix; all rotation matrices by 90, 180, and 270 degrees in all planes; scaling matrix; translation matrix and perspective matrix. The matrix-matrix and matrix-vector multiplication has been verified using the Wolfram Alpha calculator.

For the perspective, rotation, and translation matrix test, we require the tests to output known matrix values. Specifically, we test each rotation axis by 0, 90, 180 and 360 degrees (knowing 0 and 360 degrees should result in the identity matrix), and a comparison between -90 and 270 degrees. Translation matrices are tested by the application of a predefined vector and asserting the location is as expected. This includes zero translation, standard translation by positive and negative numbers, and checks for properties such as the matrix inverse by the matrix always leading to the original vector. Perspective projection, similar to the provided example, is tested by assessing each input parameter and comparing against known calculated values. This includes: the field of view; aspect ratio as well as near and far values, changing only one variable at a time. Each of these tests have been verified through own calculations carried out using a calculator according to the formulas provided in ExG3.

Task 2: 3D renderer basics

Data values	Muhammad's computer	Sameh's computer
`GL_RENDERER`	NVIDIA GeForce RTX 3060 Laptop GPU/PCIe/SSE2	NVIDIA GeForce RTX 4070/PCIe/SSE2
`GL_VENDOR`	NVIDIA Corporation	NVIDIA Corporation
`GL_VERSION`	4.3.0 NVIDIA 566.36	4.3.0 NVIDIA 555.85

Table 1: GPU vendor, renderer and version

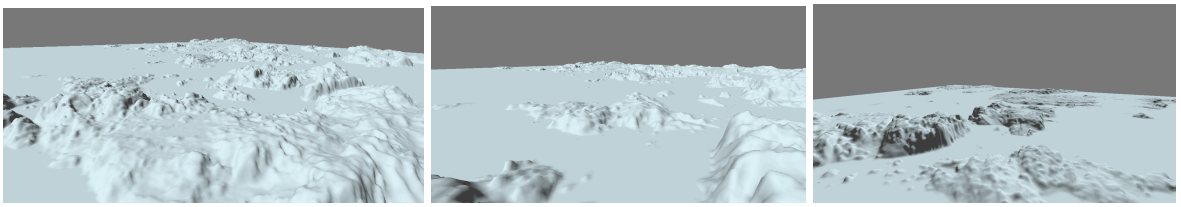

Figure 1: Loaded terrain model with applied directional lighting

Task 3: Texturing

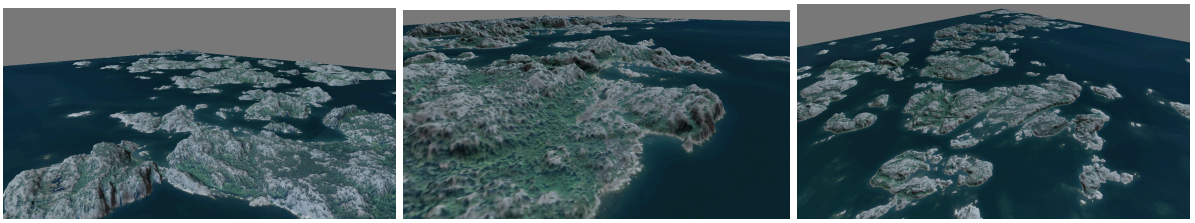

Figure 2: Textured applied to loaded terrain model

Task 4: Simple Instancing

Coordinates for instance 1: (2, 0.005, -2)

Coordinates for instance 2: (5, 0.005, -7)

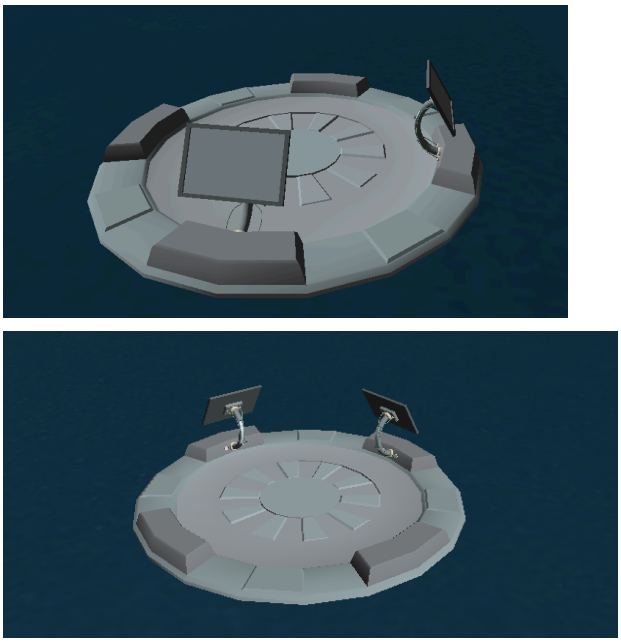

Figure 3: Two separate instances of the launchpad. Red boxes indicate location in world space

Task 1.5: Custom model

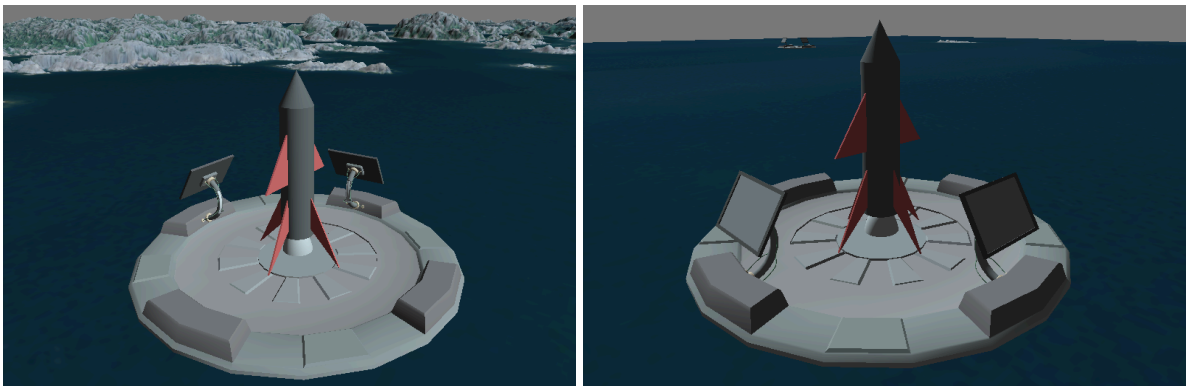

Figure 4: Rocket model

Task 1.6: Local light sources

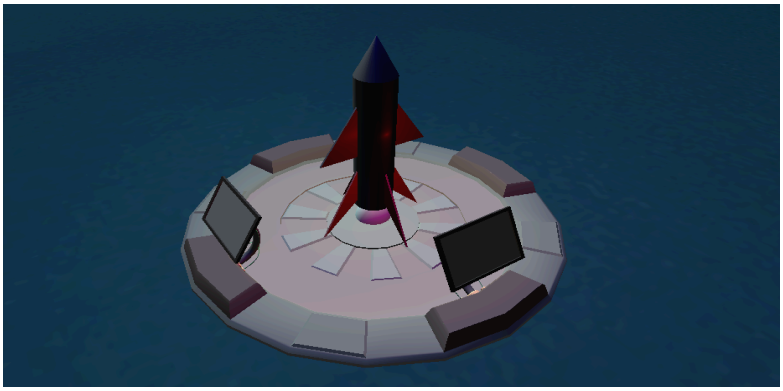

Figure 5: Rocket with point lights and Blinn-Phong shading. Green light directly behind red light on the other side of the ship. Blue light on the cone.

1.7: Animation

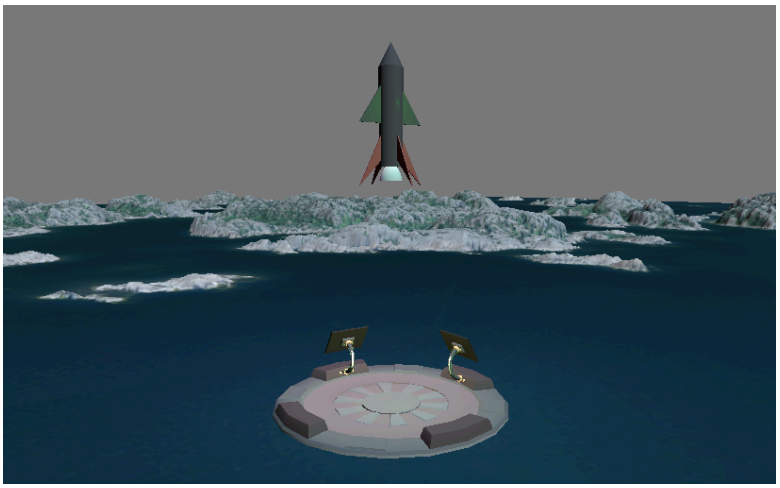

Figure 6: Rocket accelerating vertically

1.8: Tracking cameras

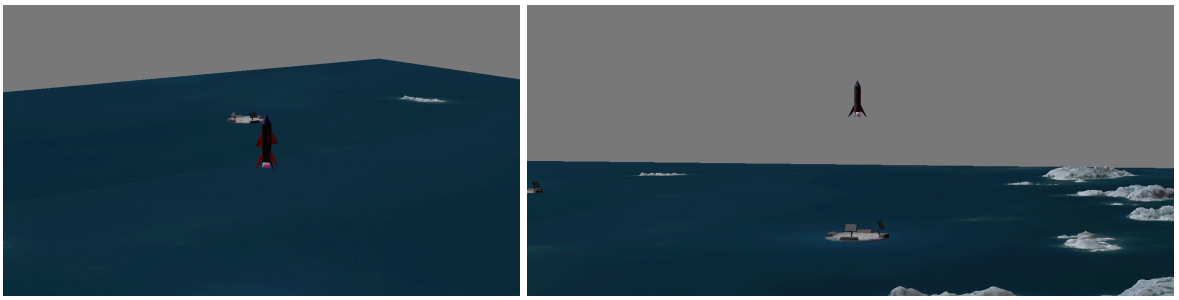

Figure 7: Chase camera (left) and ground camera (right) showing rocket mid-flight

1.9: Split screen

The split screen implementation is achieved by rendering the scene twice - each time with a different camera view and projection matrix, and then dividing the screen into two distinct viewports. The rendering of the scene twice is done using the renderScene function and the screen is split horizontally 50/50. For the top screen, a free camera was not included as it felt obsolete and clunky to have the ability to have two free cams at once.

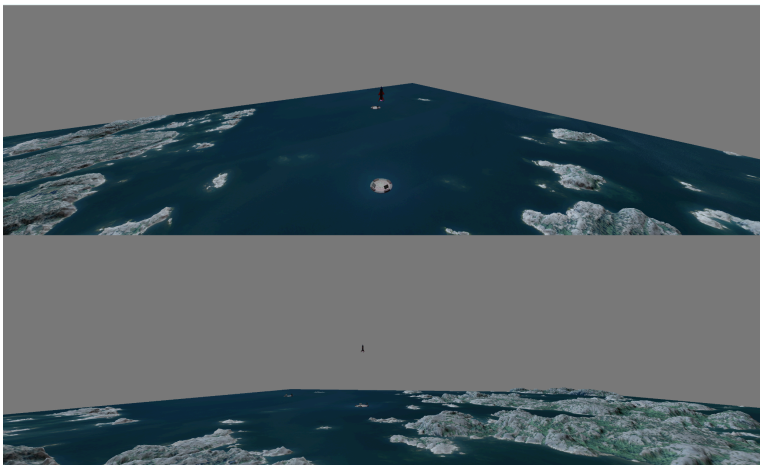

Figure 8: Split screen showing ground view (bottom) and chase view (top)

1.10: Particles

To implement the particle system, we opted to generate the particles based on point sprites. To do this, we needed to take position inside the rocket nozzle and direction on where the particles should be going. This was done by calculating a normalised vector between the centre of the rocket and the exhaust position to obtain a directional vector. The direction and position is applied as part of our rocket model, which was aligned in the x axis before rotating the rocket to a vertical orientation, and the pre-transform matrix was accounted for. The position and direction are passed through the rendering loop. When matrix transformations and rotations of the rocket are applied, we can calculate direction and position vectors to propel the particles in respect to the model.

During setup, a call to a setup function helped to set up the particle system's VBOs and VAOs that will send the particle data (discussed later on) to the custom vertex and fragment shaders tasked specifically for rendering particles and point sprites. We also load the particle textures, using the explosion.png supplied in exercise G6. This was chosen particularly for the already known alpha channels and seemingly logical particle selection for a rocket nozzle, where the explosion represented a fiery gas propelled from a typical rocket. The particle texture was loaded into the program via a function call to load an image and extract texture coords as well as RGBA values. This sprite is then passed onto the rendering phase.

Our strategy to render and save the state of the particles was to save each emitted particle to a vector, which will then be rendered during the render loop. Within this vector, we save the particle velocity, position/point, lifetime, and size. Every particle generated had some random velocity offset, and each particle lasted for 5 seconds after spawning. If the lifetime reached zero, the particle was removed from the vector and therefore was no longer rendering. As the rendering loop executes, calls to update the particles are done only if the rocket is moving. 1 particle is emitted every 0.0002 seconds to give the rocket a propulsion-like look without emitting too many particles, which may cause rendering and framerate issues.

We opted to update VBO and VAO every iteration of the render loop to ensure the particles can be calculated correctly and not offset the load to the shader programs. This is because every new render stage had dynamic variables. Writes to the shaders are performed with the ability to change the point size in the shader.

To deal with depth ordering, the particles are emitted with depth masking off during drawing. We did this to ensure particles do not disappear when accounting for transparent pixels, as some elements may become hidden behind other particles. We also enable the transparency blending function which allows particles to appear semi-transparent and blend smoothly with the background and other particles. The particles were also rendered after the opaque 3D objects to ensure particles were visible at all perspectives.

Some assumptions made during particle rendering included the idea that all rocket exhaust fumes move at a constant velocity after ejection from the nozzle. We also assume no collision between particles or surfaces to limit calculations the CPU has to make. Particular limitations of the approach is the inability for the particles to disappear after crossing past the landscape floor. Also spacing between 1 chuck of particles and the next omitted one is limited on the cycle of the rendering loop. If the cycle was faster, the spacing decreases. However, this is made up for by the particle's random movement and therefore, makes the rocket seem like it emits a continuous stream of particles from a far distance.

In terms of efficiency, the particles are rendered only for the period of time in which it is needed, with the vector throwing away timed-out particles, and particles are not emitted when the screen is minimised or being moved. Upon initial inspection, performance is good. There are no lag spikes or jittering of the rocket, particles or terrain. This can be expected given that on average 25,000 particles, and therefore points, are updated and exist on screen (relatively small list with few values to update) which are later sent to the GPU for rendering.

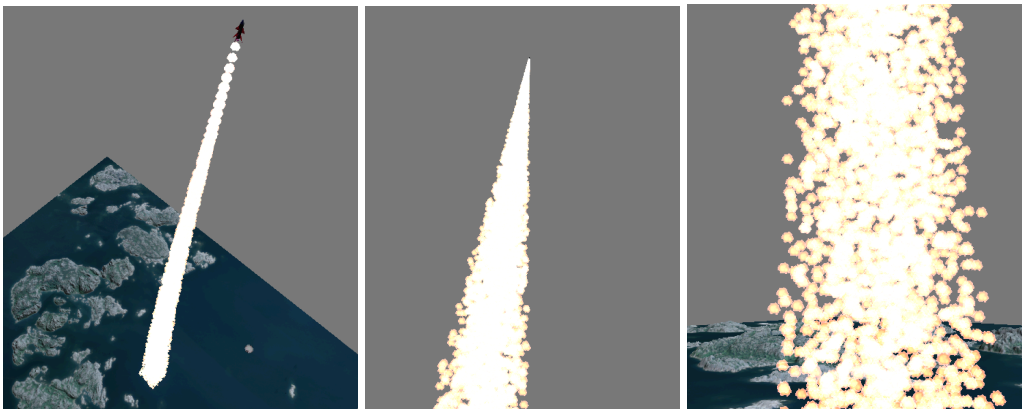

Figure 10: Rocket particle images taken from the ground (centre) and above the rocket (left). Right image shows a close up of the particle stream after ~4 seconds after lunch.

1.11: Simple 2D UI elements

To implement the text, an interpretation of fontstash's OpenGL API suited for modern OpenGL was utilised. To do this, we implemented the functions necessary to construct the backend. The functions used consists the: renderCreate (used to create a texture of given size); renderResize (resizes the texture when the atlas is modified); renderUpdate (updates texture data); renderDraw (draws font triangles with a given texture); and renderDelete. A pointer was also passed in to a custom struct which stores the VBOs and VAOs of the colours, textures and coordinates, vertexes and shader program ID. This is all initialised in the glfonsCreate function call, which sets up fontstash with the necessary parameters and objects required for rendering.

In terms of using this backend, the required custom text shaders and assigning the shader ID aided in creating the context, along with the window dimensions and alignment rules. A font was also set by passing in the fontstash context, name and asset path of the .ttf file. We can call upon the value returned, to reference this font later.

To draw text and/or buttons, we drew this to the last set of items with a glViewport using the same dimensions as the window to draw to. Text drawing was done by calling renderText with the fontstash context, text as a char*, x and y position, color as an unsigned int, and the font. Consequently, the backend renderDraw function is activated, allowing this data to be drawn to the window. renderDraw uses orthogonal projection to lock text to the screen (given window dimensions) meaning resizing will not displace, shrink or grow the text. renderText uses a pixel based system that starts with the top left of the screen being (0, 0).

To create a button, we implemented the Button class that takes normalised screen coordinates (between [0, 1]), normalised width and height, text as a string, the fontstash context, the font ID, and a custom button shader program. The class contains modifiable text colour, button colour and position values that can be accessed by editing the respective array or calls to setColors. It is also possible to link functions when the button is clicked using setOnClick. Buttons are created before the rendering loop.

When the call to create the button is made, VAOs and VBOs are instantiated and data is saved to the vertex positions such that they create 4 triangles to make 2 rectangles. One rectangle serves as the outer border and another is inside this rectangle to serve as the transparent fill (using RGBA). Button dimensions are in respect to the window dimensions so resizing will affect the size of the button. When this is complete, the vertex colours are updated according to the state they are in (either hovered, pressed or neutral). These states are updated by continuously tracking the mouse position and state on the screen during the render loop. To draw, each buttons' render function is called with window height and width supplied to then pass onto the shader to calculate the orthogonal matrix, while also sending the VAO and VBO data. Text is drawn after rendering of the button is complete. As with particles, depth testing was disabled and and blending was applied to each call to render in order to draw the combination of the 3D scene and 2D buttons together.

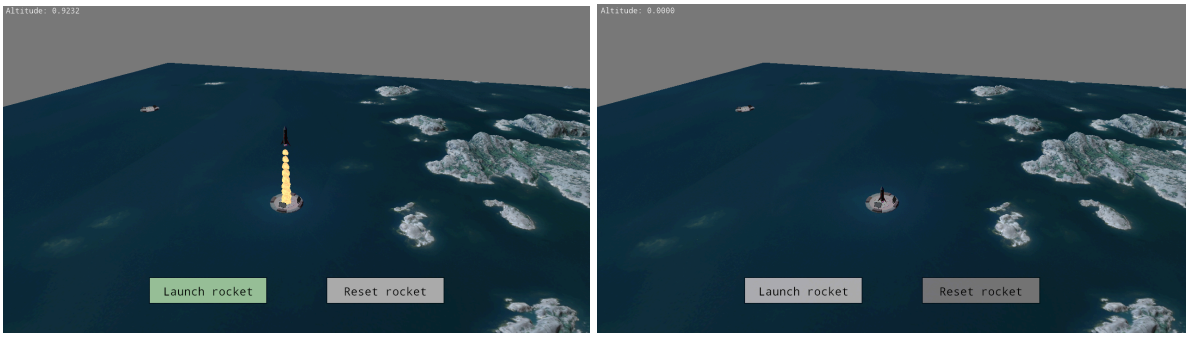

Figure 11: UI elements that include the altitude (top left of each screen), and the buttons. Buttons with a green tint indicate they are being hovered over by the mouse, and greyed out buttons indicate a pressed button.

1.12: Measuring performance

The performance tracking implementation uses OpenGL timestamp queries to measure GPU execution times for specific rendering tasks, alongside CPU timing via std::chrono to track application-level processing times. This is achieved by issuing glQueryCounter calls at the start and end of key rendering stages, such as terrain, spaceship, and sub-view rendering. The retrieved timestamps are used to calculate elapsed times in milliseconds for each task, allowing a detailed breakdown of GPU usage.

To enable seamless data logging, the performance metrics are written to a CSV file after retrieving the timing information. The CSV includes GPU timings, CPU render and frame times, as well as user input flags and camera state transitions. This design allows for a comprehensive performance profile while maintaining synchronization between GPU and CPU queries. By resetting state variables after each frame, the implementation ensures data integrity across frames.

The tracking also adapts to split-screen rendering by capturing separate metrics for both sub-views. Each sub-view is measured independently, with timestamps marking the start and end of rendering for each viewport. The flexibility of this design allows for detailed performance analysis even in complex rendering setups like split-screen mode.

The performance tracking framework is modular, enabled through the ENABLE_PERFORMANCE_METRICS preprocessor directive. This makes it easy to toggle the functionality without affecting other parts of the application.

Component	Test 1 (Static)	Test 2 (Static)	Test 3 (Static)	Test 4 (Camera Up)
GPU Metrics (ms)				
Frame Time	16.6974 ± 1.3610	16.6962 ± 1.0225	16.6975 ± 1.0220	16.6948 ± 1.2331
Terrain	4.3234 ± 1.2740	4.2555 ± 1.3298	4.2514 ± 1.3329	4.1597 ± 1.3920
Launchpad	0.0533 ± 0.0167	0.0523 ± 0.0175	0.0522 ± 0.0175	0.0516 ± 0.0188
Spaceship	0.0056 ± 0.0029	0.0055 ± 0.0029	0.0055 ± 0.0029	0.0040 ± 0.0027
View Buffer	1.74E+12 ± 1.6374	1.74E+12 ± 1.6374	1.74E+12 ± 1.6374	1.74E+12 ± 1.6374
CPU Metrics (ms)				
Render Time	0.3930 ± 0.3090	0.4630 ± 0.3926	0.4620 ± 0.3925	0.4280 ± 0.3044
Frame Time	0.4240 ± 0.3140	0.4890 ± 0.4005	0.4890 ± 0.4003	0.4600 ± 0.3072

Table 2: Results. Values shown as mean ± standard deviation. All tests ran for a sample of 1017 frames.

Our static test is based on the chase camera angle, with the rocket on the launchpad. The dynamic camera was acamera starting from spawn, looking down and rising up to see the plan view of the map

Looking at the table, we see consistency in frame GPU Time across all static and dynamic tests. There are only small variations between static tests (16.6974, 16.6962, 16.6975). Including the dynamic test, camera movement with reading 16.6948 maintains stability around the other tests' means. There is an evident variance shift between the dynamic and static runs, however this is very slight and most likely cannot constitute an indicated shift in performance when the camera moves.

With terrain rendering there is a slight decrease in terrain rendering time when the camera moves up (4.15ms vs ~4.25ms). There is also a slight increase in standard deviation with camera movement (1.39 vs ~1.32).This is to be expected given the increase in camera height to show the terrain gradually renders the whole terrain into view.

In regards to the smaller objects, the launchpad shows minimal changes in render time between test runs, however the spaceship shows a notable yet relatively negligible drop in render time (0.0040ms vs ~0.0055ms). Both spaceship and launchpad show consistent and small standard deviations due to the size of each structure in terms of vertex count compared to the terrain.

The CPU render time is more variable than the GPU metrics. Static tests 2 and 3 show higher CPU times (~0.46ms) compared to test 1 (0.39ms). Camera movement (test 4) shows slightly reduced CPU time (0.428ms). Variability can be attributed to CPU state when running the tests. The device (Muhammad's laptop) used to test may have run something in the background while tests were processed. This also doesn't take into account other key components such as the PCIe bandwidth, however we can safely assume this was not a key limiting factor as no other high intensity processes were run during testing.

The view buffers show identical values across all tests (1.74E+12), suggesting that this is a constant overhead or measurement.

Looking at figure 12, both the static and dynamic scenarios' overall performance remains stable, with frame times consistently around 16-17ms, which corresponds to approximately 60 FPS. This indicates that the baseline performance is well-optimised, with no significant differences between the static and dynamic camera tests.

Despite the stable baseline, both scenarios show occasional performance spikes (of up to 30-32ms). These spikes are more evident in the static test, especially in the early frames. Both tests also show some drops in frame times (dropping to below 10ms at most) which could be a result of the rendering conditions used or system-level factors. When comparing stability, the dynamic camera test shows slightly more frequent but smaller variations, while the static test has fewer but more extreme outliers. This is consistent with earlier statistical data, where the static test demonstrated a marginally lower standard deviation.

There are also notable patterns in the frame time graphs. Both scenarios show some volatility in the initial frames, likely related to initialisation processes. Towards the end of the tests, there is a significant drop in performance, which could be due to cleanup or test termination routines. Also, the dynamic camera movement does not significantly impact overall frame time stability, suggesting that the renderer is well-optimised for handling camera movements.

In conclusion, the data demonstrates that the renderer performs consistently across both static and dynamic camera scenarios, with only minor variations. The slightly higher variability in the dynamic test is expected due to the changing view conditions, while the spikes in the static test may require further investigation. Overall, the results indicate a robust and well-optimised rendering system.

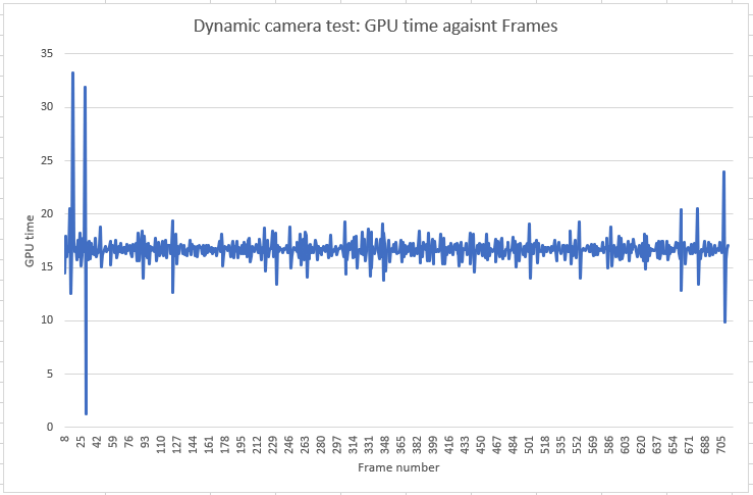

Figure 12: Static (top) and dynamic (bottom) camera test results plotted as frame number against GPU time (given as ms).

Task	Person(s)
1.1: Matrix/Vector Functions	Muhammad
1.2: 3D renderer basics	Muhammad
1.3: Texturing	Muhammad
1.4: Simple Instancing	Muhammad
1.5: Custom model	Muhammad, Sameh
1.6: Local light sources	Muhammad
1.7: Animation	Muhammad, Sameh
1.8: Tracking cameras	Sameh
1.9: Split screen	Sameh
1.10: Particles	Muhammad, Sameh
1.11: Simple 2D UI elements	Muhammad
1.12: Measuring performance	Sameh