Computer Graphics CWK2 - COMP3811

1.1 Matrix/vector functions

Firstly, we added tests to check that the matrix-by-matrix multiplication was functioning correctly. The first test multiplied two identical matrices and checked that the resultant matrix was equivalent to the mathematically correct one. The second test multiplied a matrix by an identity matrix and checked that the resultant matrix was equivalent to the original matrix. By applying an additional identity matrix test, we can ensure that the matrix is not manipulated, therefore reinforcing mathematical validity. Finally, we have added a boundary matrix case with larger values to validate that operations would also work in extremities of the world coordinate space.

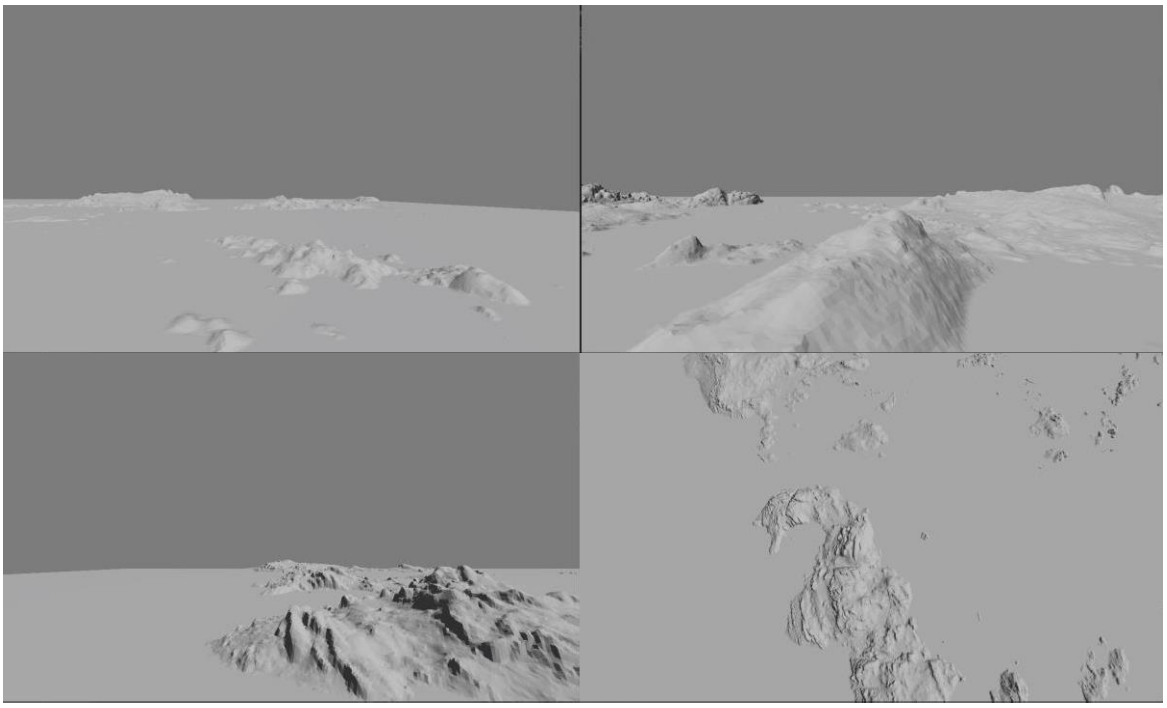
The next tests we added checked the translation function. This checked that an input vector when multiplied by a matrix created with the make_translation function correctly translated the vector by the values given to the make_translation function. This is a particularly important test given that future tasks will require valid translation through matrix multiplication, such as object placement for the spaceship. Our tests are therefore again designed to ensure that both typical and boundary scenarios are correctly computed, using large values alongside standard ones for various translations.

The next tests added were regarding matrix by vector multiplication. These tests checked that when a matrix is multiplied by a vector, the mathematically correct vector is produced. Multiplying matrices with vectors enables linear transformations which are fundamental for rendering. We test for both boundary cases, standard cases, and identity matrix cases. This allows us to cover several edge cases for assurance of accuracy.

The next tests added were regarding rotation. For each type of rotation, we added tests to determine if the correct rotation matrix is produced depending on the input values. We have included tests for zero-degree rotations and standard rotations (such as 90 degrees). For more interesting tests, we have also included negative degree rotations to test opposing rotation direction, and non-standard degree rotations. Rotation will be necessary for certain transformations, such as camera orientation and world to camera transformations, and is therefore important.

For the make perspective projection, we used a test from exercise 3 which included a field of view of 60 and a window size of 1280x720. For less standard cases, we added a test using a field of view of 70 degrees, with a non-standard window size of 1440x2560. By doing so, alternate aspect ratios are also tested which will be necessary for window size changes by the user. Finally, we added a test with a larger atypical field of view at 350 degrees. This allows us to test the limits of perspective projection.

1.2 3D renderer basics

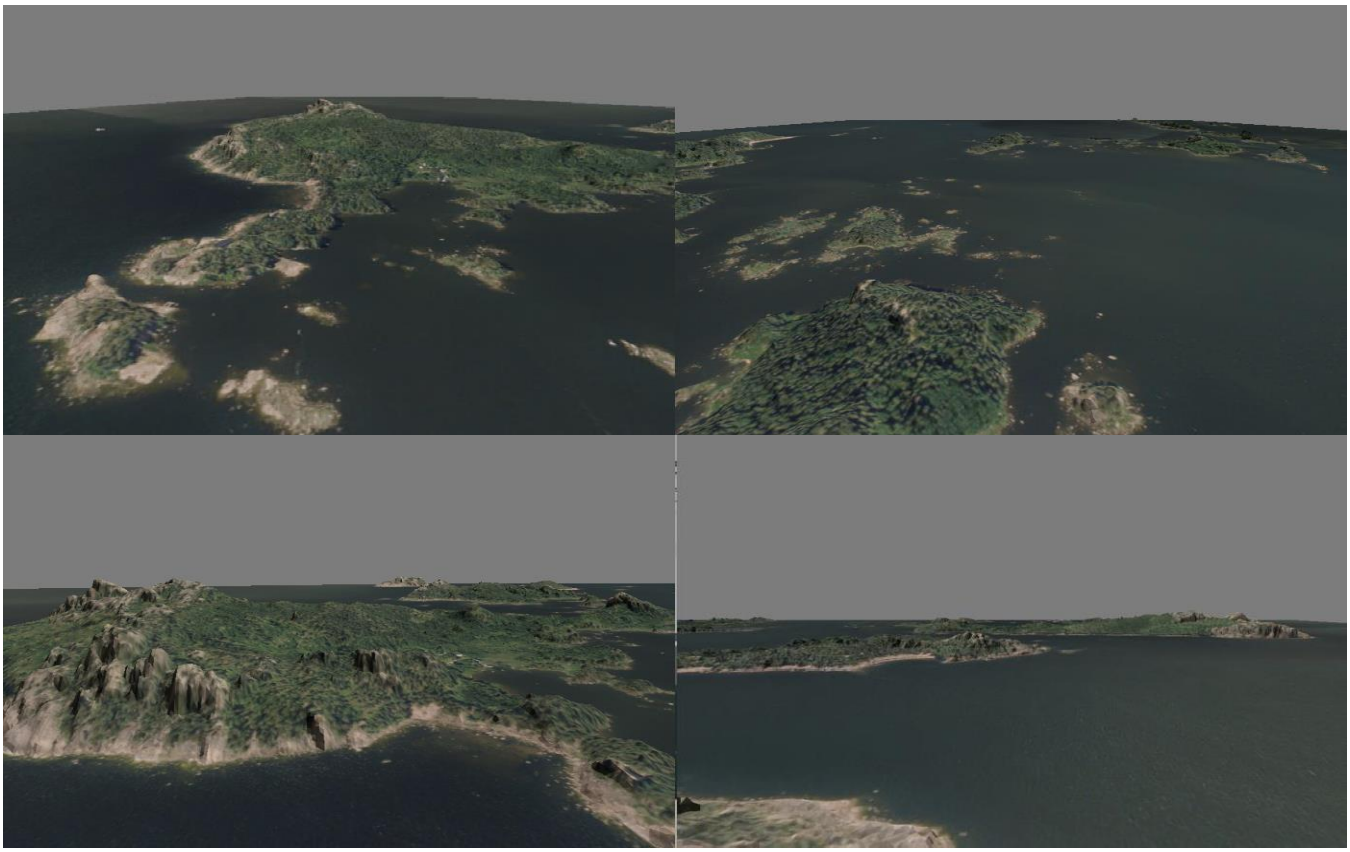


Test Computer 1:
RENDERER NVIDIA GeForce RTX 3060 Laptop GPU/PCIe/SSE2
VENDOR NVIDIA Corporation
VERSION 4.3.0 NVIDIA 531.97

Test Computer 2:
RENDERER NVIDIA GeForce GTX 1650 Ti/PCIe/SSE2
VENDOR NVIDIA Corporation
VERSION 4.3.0 NVIDIA 536.67

Test Computer 3:
RENDERER NVIDIA GeForce RTX 4070/PCIe/SSE2
VENDOR NVIDIA Corporation
VERSION 4.3.0 NVIDIA 545.23.08

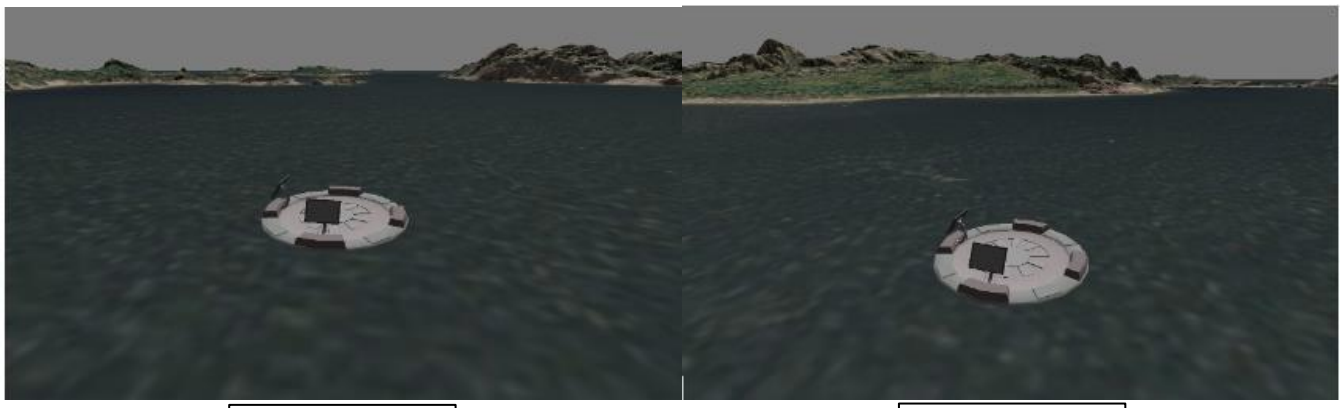
1.3 Texturing



1.4 Simple Instancing

The first launchpad was placed at the coordinates: 0.f, -0.97f, 50.f.

The second launchpad was placed at the coordinates: -20.f, -0.97f, 70.f

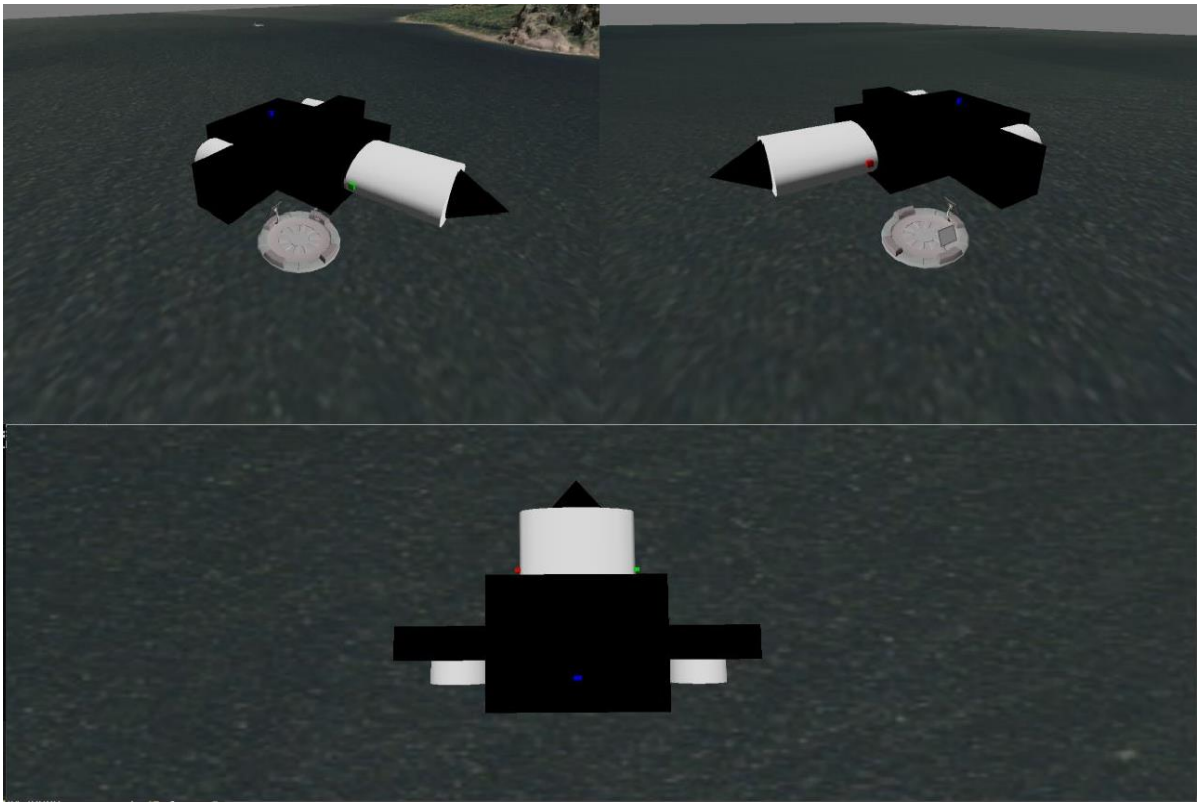


First pad

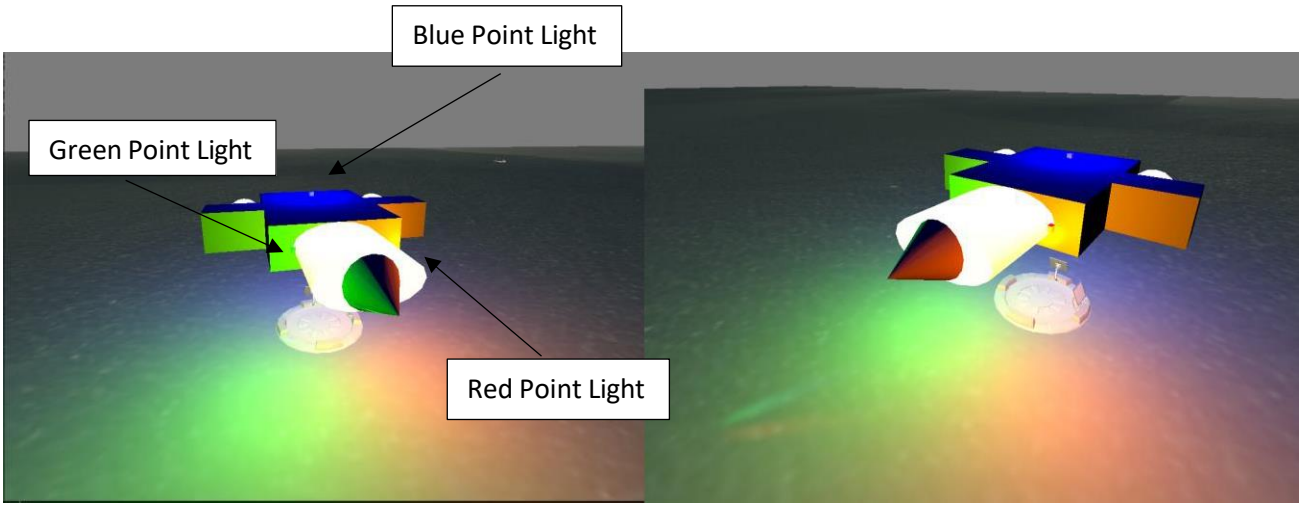
Second pad

1.5 Custom model

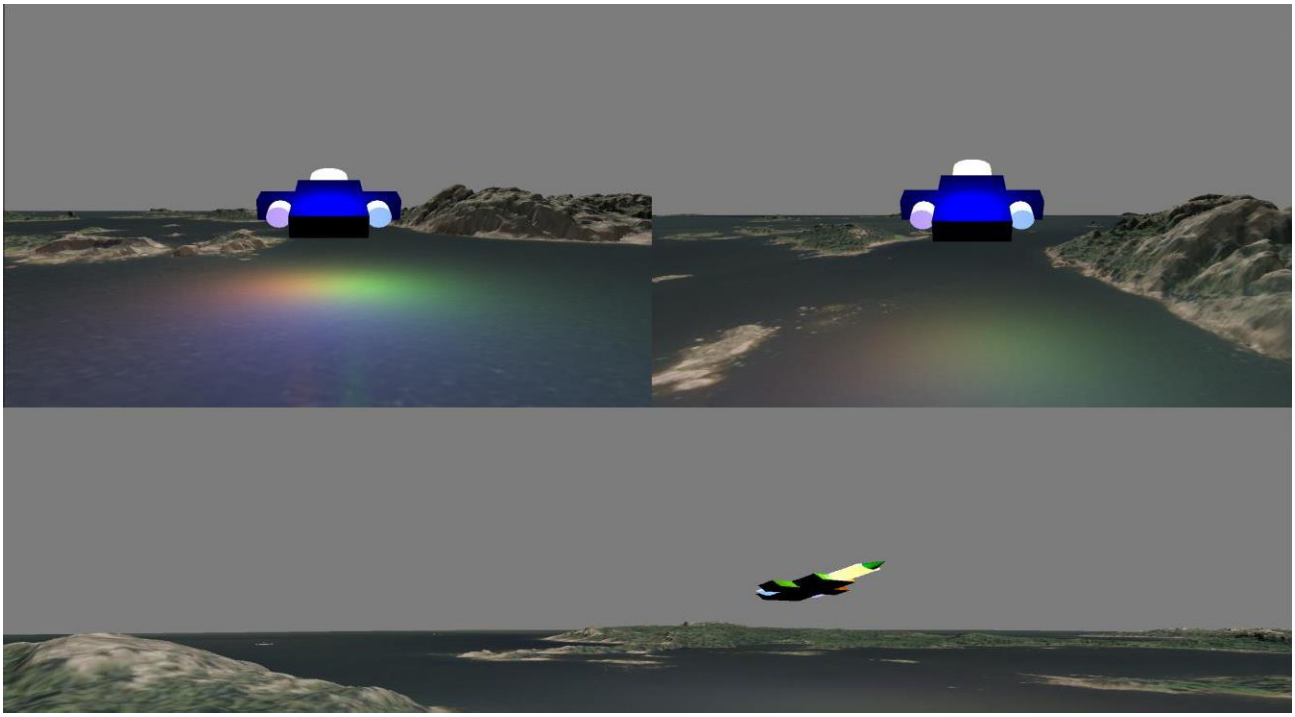
The space vehicle was placed on the launch pad at the coordinates: 30, -0.5, -20.



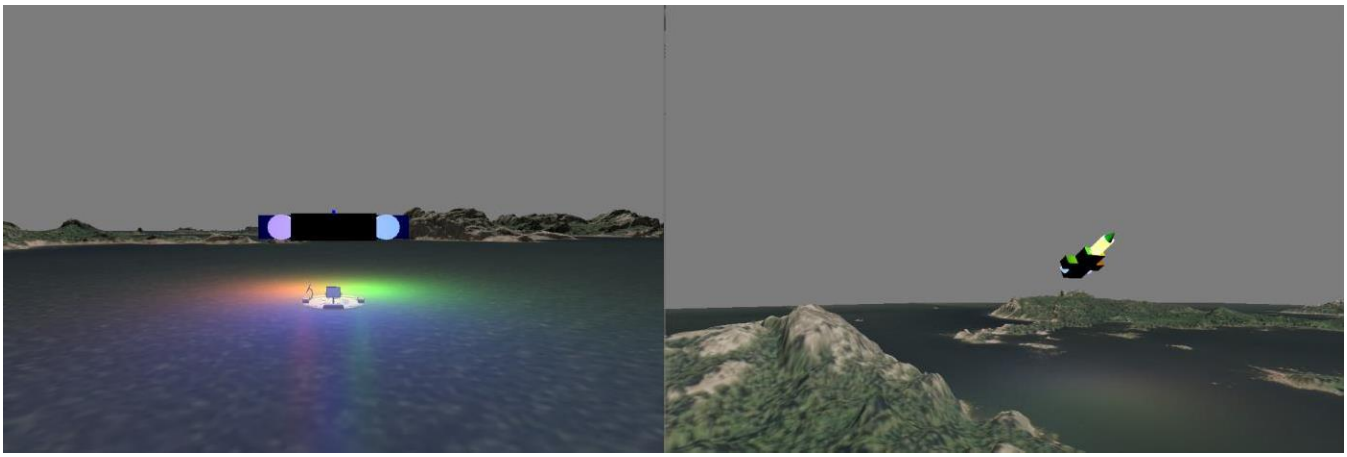
1.6 Local light sources



1.7 Animation



1.8 Tracking cameras



Fixed Distance

Ground Fixed

1.9 Split screen

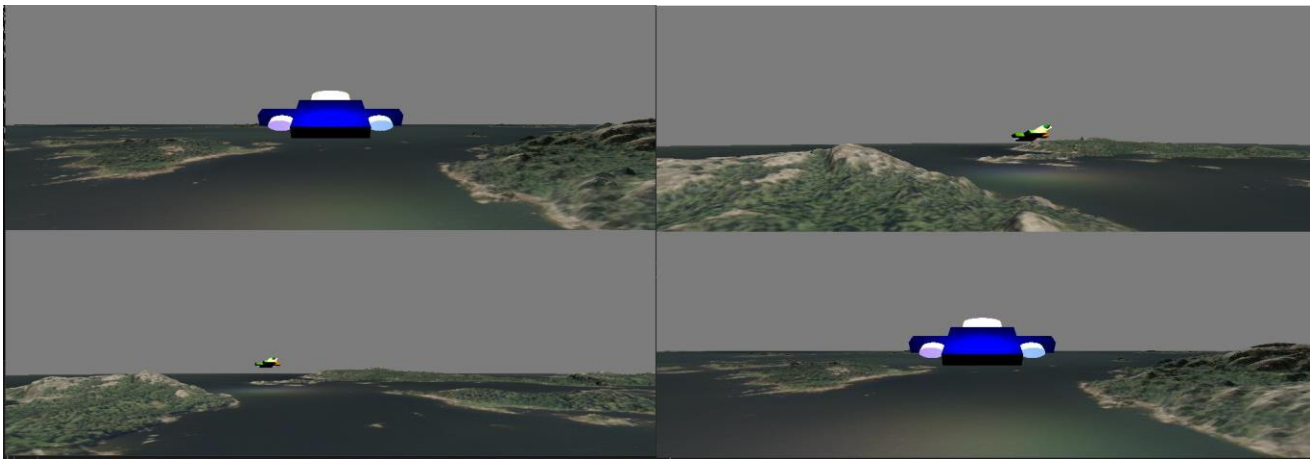
We started changing *glfw_callback_key_* to add some toggle keys of screen mode and tracking camera functions. Split screen mode can be defined as a boolean type to distinguish between ‘on’ and ‘off’. If

the ‘V’ key is pressed, the *splitScreenMode* variable is set to *true,* which will be checked in an *if* statement for rendering different viewports.

Although this allows for differently sized screens to be displayed, achieving simultaneous varying perspectives requires a higher level of sophistication. We therefore decided to use an additional camera with a separate projection and normal matrix. This camera separately renders perspectives on the lower screen when split screen mode has been enabled and is required for different world to camera projections. Performance impacts of such a change are likely to be significant, given that double the number of objects and viewpoints are rendered. However, we make use of reusable data between the viewpoints where possible, thus reducing potential memory expenditure.

Multiple *glviewports* in the main loop are in charge of showing different views in accordance with user-triggered parameters. For managing these keys, we decided to make another file called ‘utilities’ consisting of a header and source file. The *camera_switching* method in the ‘utilities’ file contains different camera modes for each screen, depending on the split-screen mode.

When setting FixedDistance mode, the camera position is directed towards our space vehicle with a constant dimensional value to achieve a fixed distance. Our ground fixed mode defines a constant position using ground coordinates, ensuring the camera does not change position. However, the camera orientation is directed towards the vehicle by applying trigonometric arctan functions to camera-vehicle vectors, resulting in phi and theta angles for rotating the camera.



1.10 Particles

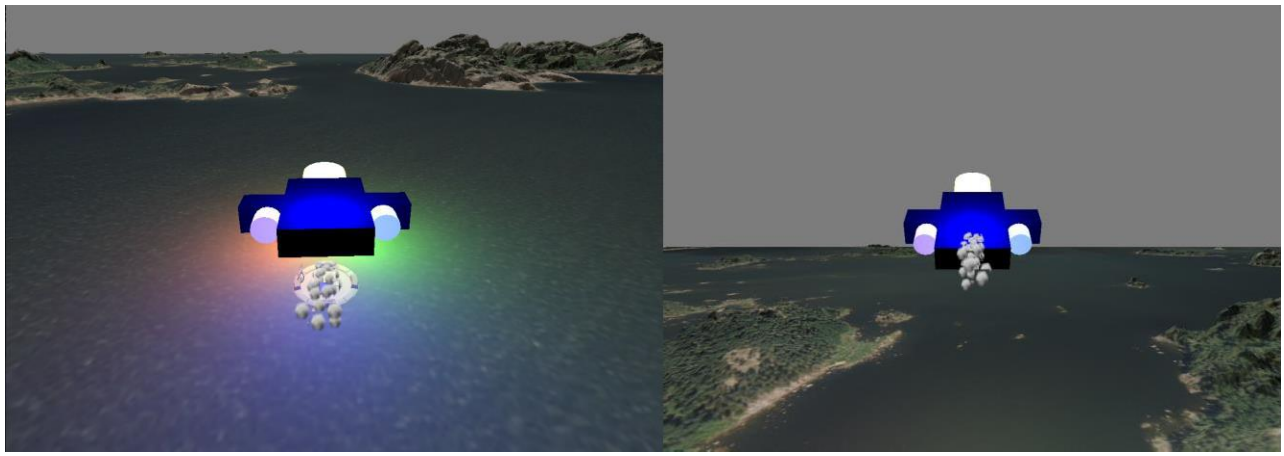
The particle system has been implemented primarily via instancing for efficient memory utilization and processing. We have included an additional vertex and fragment shader capable of handling transparent textures, alongside the instances for each particle. The system operates via a new ‘ParticleSystem’ class which allows for animation updates and initialization. On construction, the class samples from a normal random distribution for velocities, starting positions, and time to live. These values remain fixed from construction onwards. Although applying randomization for the properties of particles at rebirth would result in a more realistic simulation, randomization can be CPU intensive. We therefore decided to

choose a less demanding approach by fixing the initially randomized attributes of each particle, throughout the duration of the system. This leads to more efficient utilization of the CPU.

The number of particles remains fixed through exploiting a time to live system. Each particle is initially assigned its time to live which determines how long it can travel for before being repositioned at its initial starting position (starting position is relative to the spaceship exhaust). This may allow for a reduction in memory utilization and no dynamic allocation but also restricts the system. The number of particles must be fixed at compile time and cannot change based on events within the world. However, the system is still flexible in how it may be rendered including changing the density (velocity distribution variance), length (time to live variance), particle size (cube mesh scaling), and more.

The billboarding effect (see screenshots below) has been achieved via an inversion of the camera orientation matrix, which is applied at vertex level to each particle. This allows for a rotation of the particles so that they remain oriented towards the viewer, no matter how the camera is positioned. Although inverting matrices can be intensive, it is performed only once per frame. Particles are translated to the origin for rotation such that they remain fixed in their desired position. The limitation of this effect is the added complexity of additionally directing the particles away from the vehicle. In our system, this was not fully achieved. A matrix translation of each particle's velocity with the normal of the spaceship's direction was applied, but the results were not as expected.

For transparency, the particles have been textured and rendered using additive blending and an alpha channel.



1.12 Measuring performance

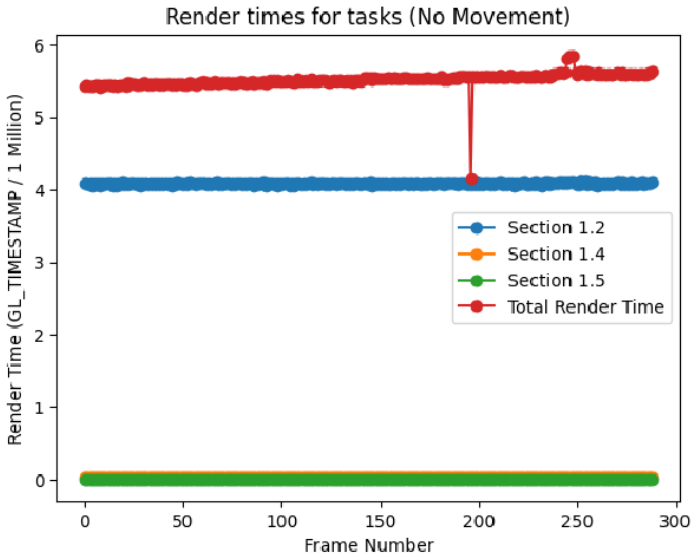
Performance of the system was measured through GL_TIMESTAMP queries, alongside CPU timers. To minimize waiting/stalling, our benchmarking system applies a busy-waiting approach. This involves a separate query to determine whether the GPU has finished for a given timestamp request 'GL_QUERY_RESULT_AVAILABLE'. The query is blocking and continually polls until a result is returned. In terms of waiting times, using this approach is faster, given that querying for whether a result has been obtained is instantaneous. Therefore, as soon as the result becomes available, rendering continues.

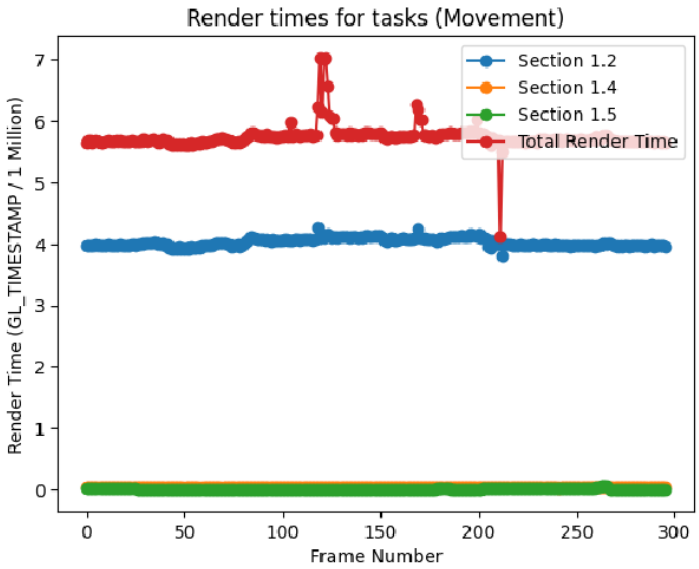
As shown below, the results have been obtained for render times, with and without movement. Benchmarking was performed using an NVIDIA RTX 4070 (test computer 3) over 300 frames. From the data, we can firstly see that section 1.2 takes the longest time to render out of all measured sections. Given the relative complexity and size of the terrain object in comparison to section 1.4 and 1.5, this is unsurprising. The vertex and fragment shaders must compute interpolation and colouring over a much larger number of vertices and pixels, and this results in more extensive overhead. This is further reinforced by the insignificant render times of section 1.5 and 1.4. The spaceship and landing pads are somewhat similar in terms of complexity and size which corresponds well to their respective rendering times.

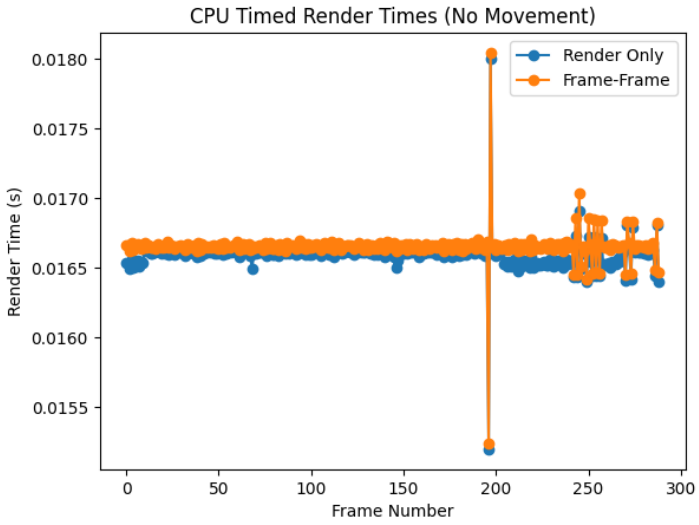
By comparing the moving and non-moving graphs for GPU rendering times, the fluctuation in performance is evident. This may indicate that certain parts of the world are more intensive for the GPU, such as Blinn-Phong shading, where several extra computations associated with the point light vectors and directions must be taken into account. This will include matrix multiplication, interpolation, and data retrieval. Interestingly, total render time falls to the same time as section 1.2, for at least 1 frame in both the moving and non-moving tests, at the same time. This may indicate some deterministic occurrence within the rendering process that repeats periodically but may also be anomalous.

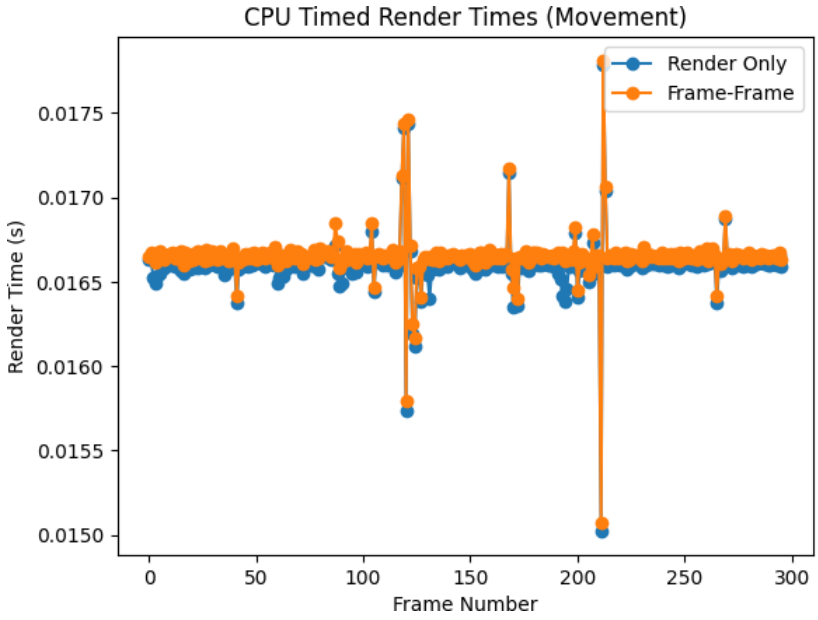
Moving on to the CPU timed benchmarks, one of the more evident occurrences is the similarity between render times and frame-to-frame times. This indicates that rendering commands make up the largest proportion of computation time, i.e., the GPU is much more heavily utilized in rendering than that of the CPU. Most non-rendering commands will be performed using the CPU and is therefore expected. The anomalous frame times, as discussed previously, also occur again and correlate with the previous graphs. However, we now see that the subsequent CPU timed render times for the next frame are much larger than usual. The rendering pipeline may potentially be accounting for something that occurred within the previous frame. The fluctuation, as imposed by camera movement, is also evident in CPU timed benchmarks, reflecting the changing rendering times for each of the objects.

Overall, from benchmarking we can conclude that changing our camera position and viewpoint will notably impact the performance of the system. This implies both that objects and data are only rendered, when necessary (within the viewport), and have various computational demands. We have seen the extent to which GPU computations (rendering times) take over the rendering pipeline, and potential anomalous/unexpected data values relating to atypical infrequent scenarios when rendering.


Render times for tasks (No Movement)
Render Time (GL_TIMESTAMP / 1 Million)
Frame Number
Section 1.2
Section 1.4
Section 1.5
Total Render Time


Render times for tasks (Movement)
Render Time (GL_TIMESTAMP / 1 Million)
Frame Number
Section 1.2
Section 1.4
Section 1.5
Total Render Time


CPU Timed Render Times (No Movement)
Render Time (s)
Frame Number
Render Only
Frame-Frame



Appendix

	Contribution	What code
Savan Dosanjh [sc21srsd]	1.2. 3D renderer basic, 1.3. Texturing, 1.4. Simple instancing, 1.12 Measuring performance	Modularising code, including building utility file. Benchmarking system for rendering times.
Jakub Kurasz [sc21jk]	1.5. Custom model, 1.1 Matrix/vector functions	Combination of methods to create shapes for building the customised spaceship. Matrix/vertex test functions
Hanmun Hwang [sc21h2h]	1.6. Local light sources, 1.7. Animation, 1.8. Tracking cameras, 1.9. Split Screen, 1.10 Particles	Building shader programs for lighting and particle systems. Integration of switching camera with split screen in utility file, adding camera transition key inthe callback method.