1.1 Matrix/ Vector functions

(*) operator Matrix x Matrix

To do this I used nested loops to iterate over the matrix and do the sums required putting the new value into a results matrix. To test this I multiplied by the identity matrix and in another test compared it with a random matrix to ensure it gives the answer i have pre calculated.

(*) operator Matrix x Vector

To do this I calculate each row and then put it into the return as a vector. To test this I multiplied it by a known vector a couple of times and checked it against a pre calculated answer.

Rotation (x, y, z)

To do this I changed only the 4 values in each matrix corresponding to a rotation. To test this I checked x,y, z rotations against a known rotation (70 degrees) that I pre calculated.

Translation

To do this I added to the matrix the x, y ,z of the vector. To test this I translated a matrix with a random pre calculated translation and checked it was correct.

Perspective projection

To do this I followed the guide in G3 to make a prospective projection matrix. To test this I kept pre-made tests for this.

Scaling

To do this I set the scale attributes using the inputted matrix. To test this i checked it against a pre known scaled matrix.

1.2 3D render basics

Values of for test computers

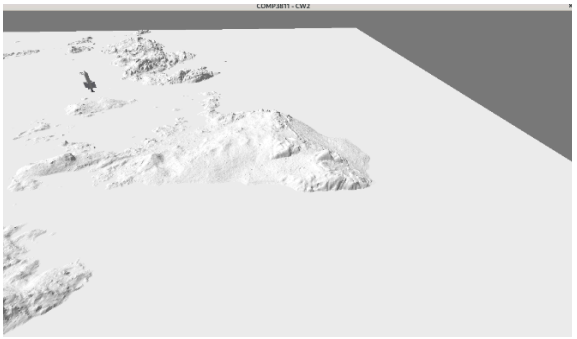
Jake's laptop

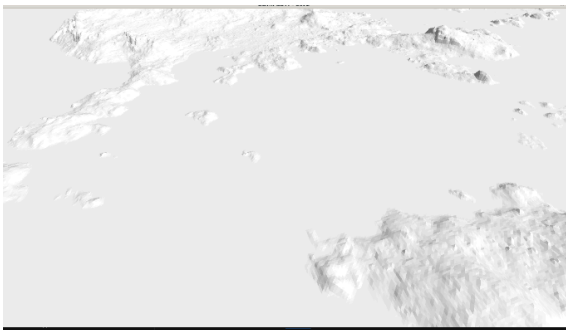
RENDERER AMD Radeon(TM) Graphics
VENDOR ATI Technologies Inc.
VERSION 4.3.0 Core Profile Context 23.10.24.02.230811
SHADING_LANGUAGE_VERSION 4.60

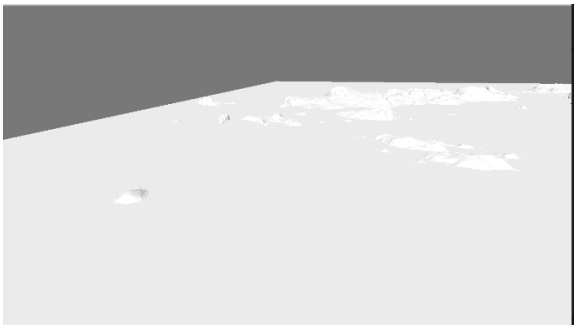
Lab UG Computing lab computer type 1

RENDERER Mesa Intel(R) UHD Graphics 630 (CML GT2)
VENDOR Intel
VERSION 4.6 (Core Profile) Mesa 24.2.8
SHADING_LANGUAGE_VERSION 4.60

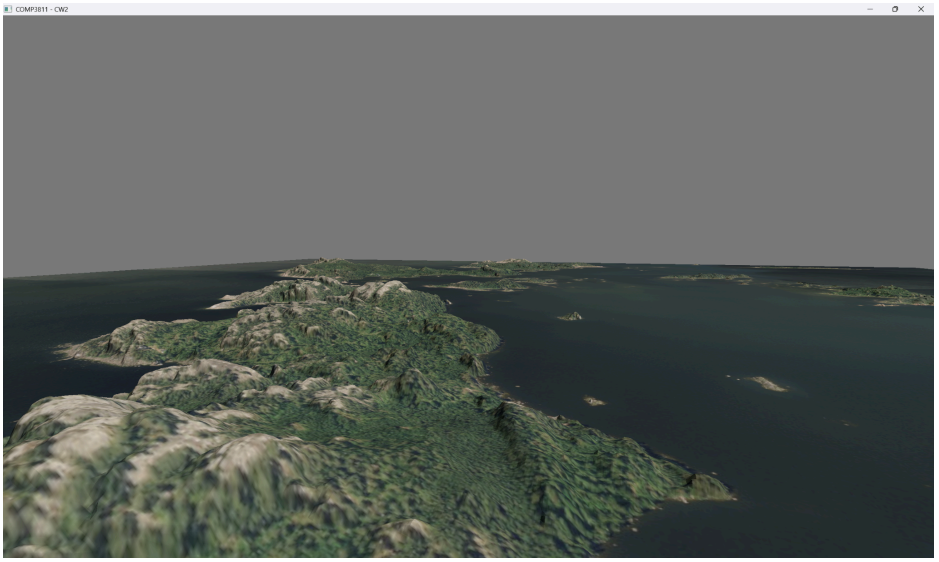
Screenshots of screen

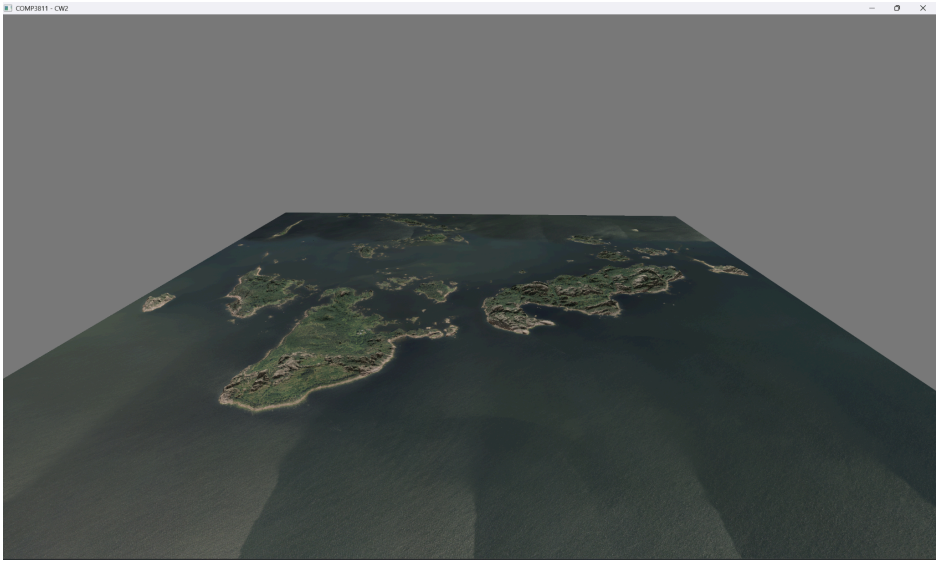


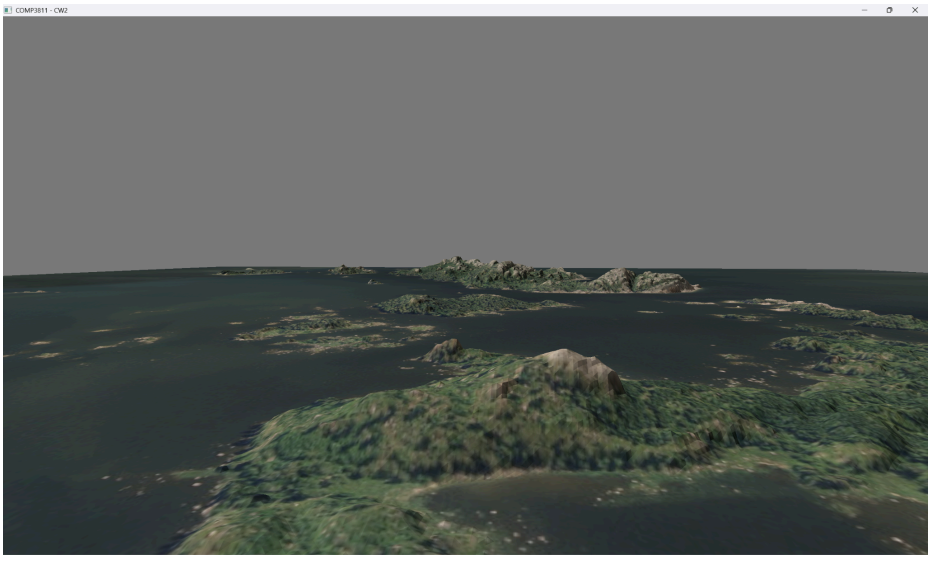




1.3 Texturing





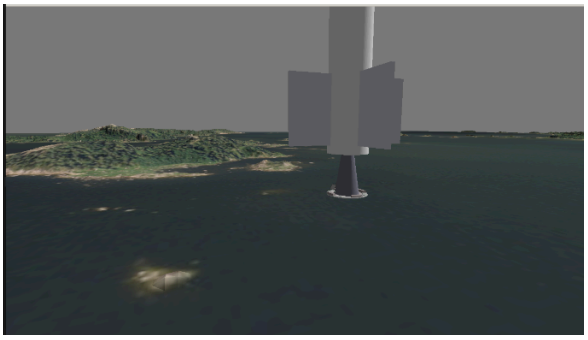


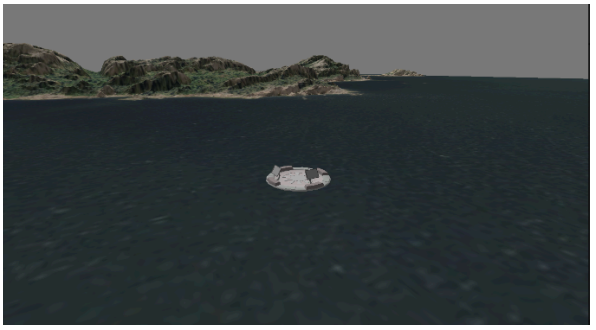
1.4 Simple instancing

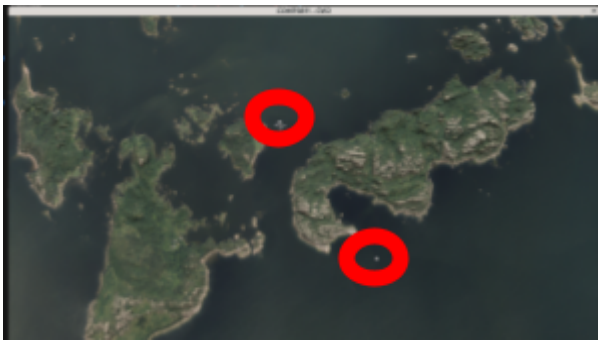
Coordinates

```
Vec3f landingPadPosition1 = {30.f, -0.95f, 30.f};
Vec3f landingPadPosition2 = {0.f, -0.95f, -5.f};
```

Screenshots



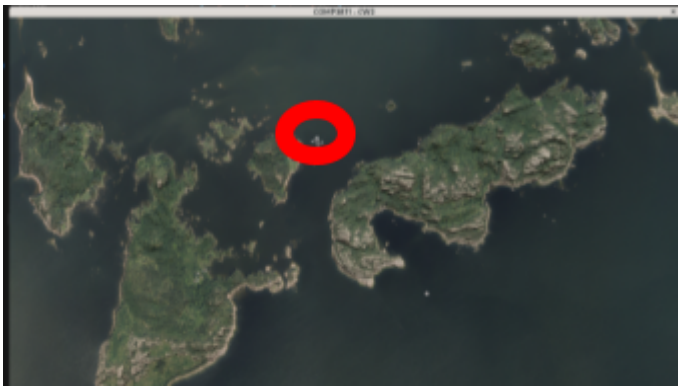


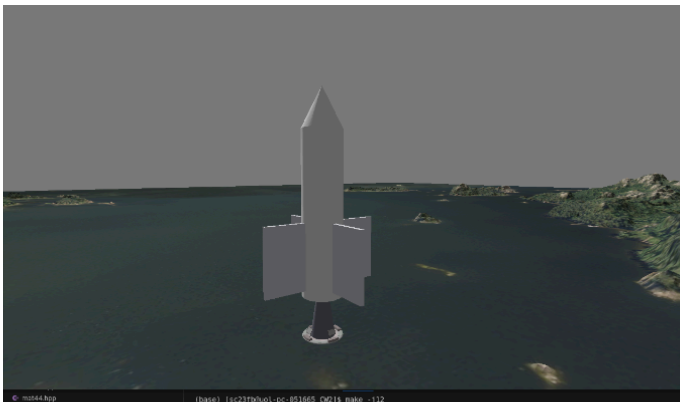


1.5 custom model

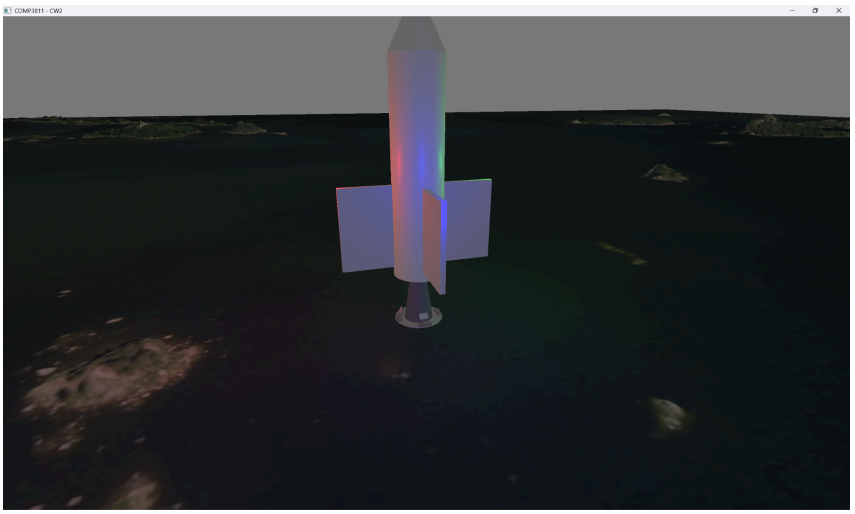
Where is the rocket

Vec3f landingPadPosition2 = {0.f, -0.95f, -5.f};

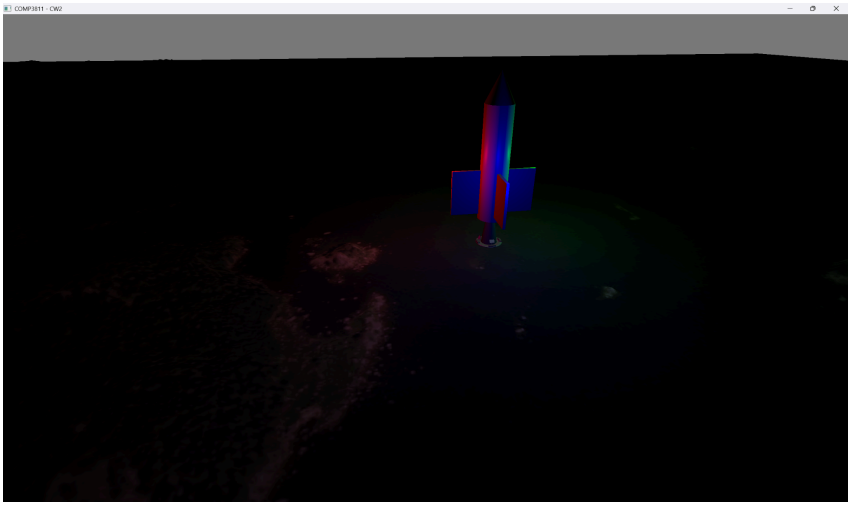




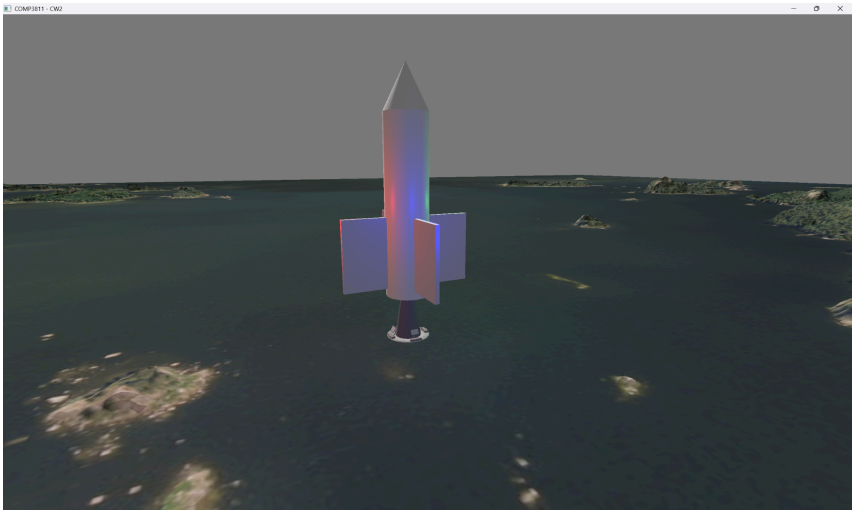
1.6 local lights sources



Rocket and landing pad with global directional lighting disabled and all 3 local lights enabled



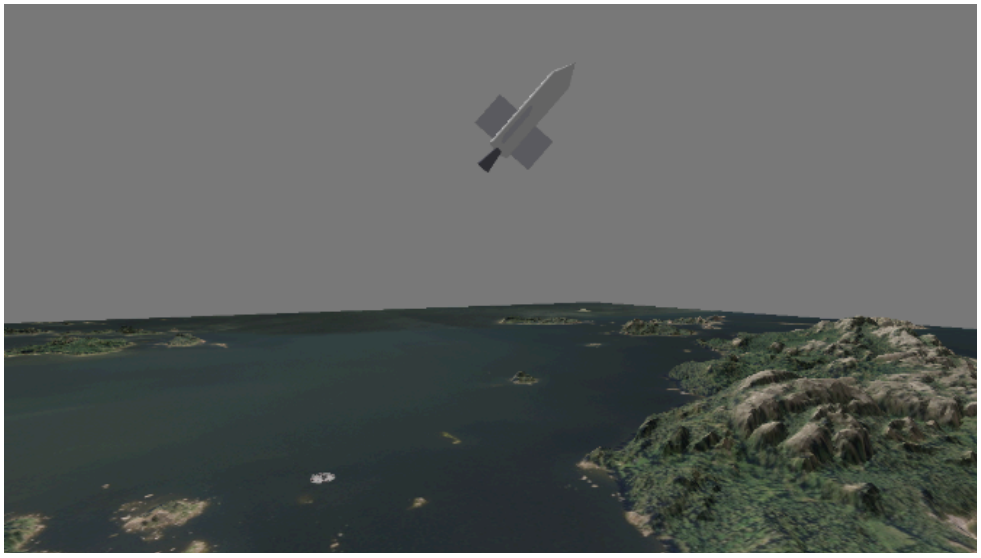
Rocket and landing pad with global directional lights and ambient lighting disabled and all 3 local lights enabled



Rocket and landing pad with global directional lights and local lights enabled

1.7 Animation

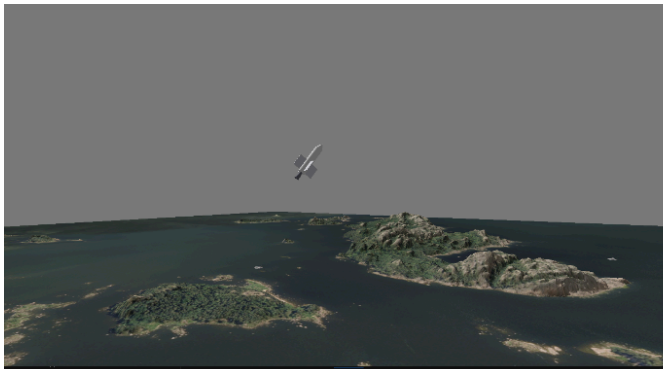
Screenshots of rocket



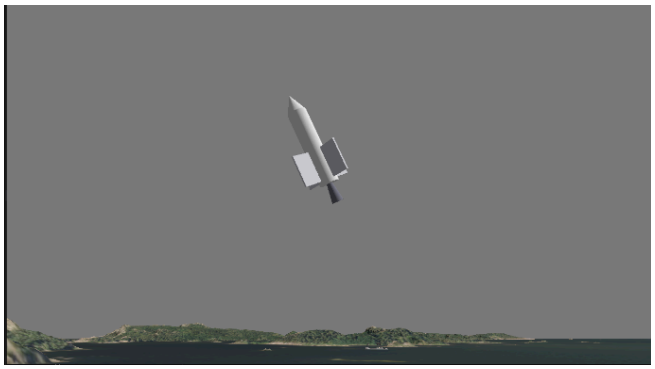
1.8 Tracking cameras

Screenshots

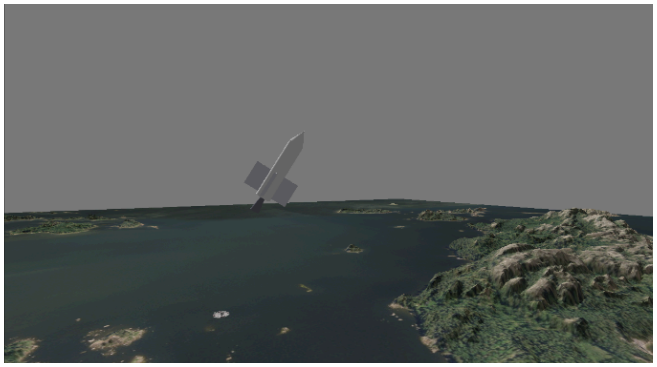
Free



Ground



Fly with

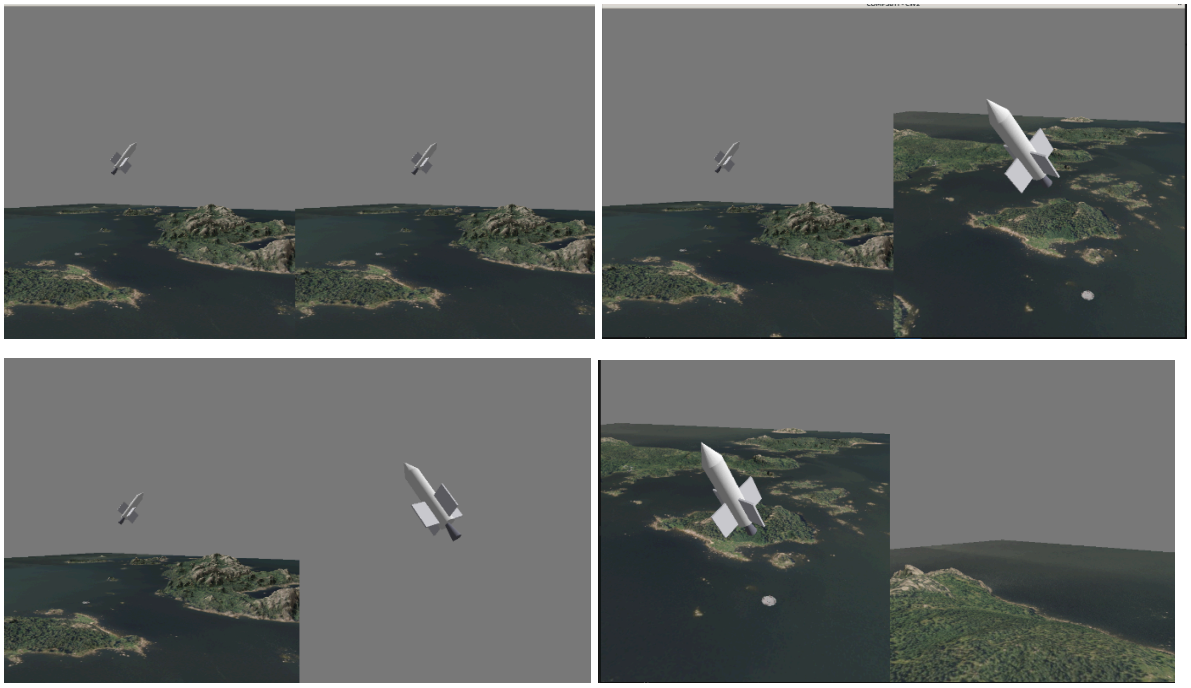


1.9 Split screen

How was it implemented?

I implemented split screen by allowing the user to toggle it on and off with the V key. When it is enabled I run the drawing calls for the scene a second time from the 2nd perspective utilising glViewPort (https://learn.microsoft.com/en-us/windows/win32/opengl/glviewport) to display the camera views side by side. The downside to this is that all the geometry has to be rerendered from a 2nd perspective effectively doubling the work required.

Screenshots



1.10 Particles

The particle system is run on the CPU with a fixed size pool of particles where each particle has a velocity and a lifetime, once the lifetime of a particle goes below 0 it is recycled (i.e. sent to a new position and given a new lifetime and velocity). The velocity and lifetime are changed with relation to delta time on each frame which keeps the simulation frame rate independent. The rendering utilises OpenGL point sprites which is better than quads due to only needing one vertex per particle, textures are mapped to the vertex using built in “gl_PointCoord” UV. Depth writing was disabled in the particle draw pass to stop particles occluding each other while maintaining occlusion behind objects (terrain, rocket etc). The particles render with additive blending so particle draw order is irrelevant and is well suited for rocket exhaust (although this may not be well suited for other needs). No dynamic allocations take place as the pool is a fixed size and the GPU data is also pre allocated so no memory is allocated or freed during the render loop. Only active particles are sent to the GPU which saves bandwidth. And pooling means that dead particles are recycled rather than “instantiating” new particles. The downsides of my implementation is the lack of collision with objects, the extremely simple physics and the additive blending only approach which would not be suitable for a smoke cloud for example.

1.11 Simple UI elements

Not implemented

1.12 Measuring performance

	Terrain (ms)	Landing Pads (ms)	Rocket (ms)	CPU Frame (ms)
Initial Position	4.1	0.035	0.014	16.5
Looking at sky	2.3	0.012	0.003	16.5
Entire terrain in view	12.6	0.004	0.007	16.8
Far from rocket	4.7	0.002	0.008	17.1
Close to rocket	1.95	0.013	0.426	16.1

I used OpenGL query objects which allowed me to get timestamps at specific locations in the code. Timestamps were measured at the start of a frame, after terrain drawing, after landing pads drawing, after rocket drawing and at the end of the frame. To prevent the CPU from stalling while waiting for the GPU the CPU would only check if the measurement data was available from the previous frame and if it wasn't available yet it would be skipped. CPU frame time was measured differently instead using the standard library high resolution clock for timestamps. All the code relating to taking measurements was wrapped within "if defined" statements to allow measurements to be toggled on/off. The above data was captured on "Jake's laptop" with the specifications on the OpenGL version available earlier in the report and the rendering card being AMD Radeon Integrated Graphics on a Ryzen 5 6600HS.

The results align with the expectations of scene complexity when taking into account frustum culling. The CPU time stayed steady due to VSync being enabled which means the applications attempts to keep the frame rate at the refresh of the monitor (60HZ in this case) and will wait before rendering frames if the FPS is higher than this.

The most impactful object on performance was terrain which is expected due to its relatively high complexity compared to other objects in the scene. The data suggests that if Vsync was disabled an FPS of ~150 - 200 would be achieved.

The measurements were consistent between runs provided no other applications were open.

Looking at the sky would mean that no geometry is in view so theoretically no geometry should be rendered due to frustum culling which aligns with it being the most performant run. Being closer to an object (such as the rocket) means it takes up more of the screen and thus takes up a larger percentage of the render time. Viewing the whole terrain at once causes the entire geometry to be rendered which will incur a steep performance cost which is shown in the table. These extremes are not often encountered in nominal viewing situations.

Finally, using split screen will theoretically double the amount of rendering work required, however the implementation only measures one camera view so we do not have data for split screen.

Appendix

Task	Done by
1.1	Finlay
1.2	Finlay
1.3	Jake
1.4	Finlay
1.5	Finlay
1.6	Jake
1.7	Finlay
1.8	Finlay
1.9	Finlay
1.10	Jake
1.11	Jake
1.12	Jake