Aodhan Gallagher - sc20ag | Kieran Cunningham - sc21khc | Jibran Raja - ed20j3r

COMP3811 Coursework 2 - Report

1.1 Matrix / Vector Functions

We began the coursework by implementing a series of matrix and vector functions in order for our program to function correctly. The functions we implemented included: Mat44f operator*, Vec4f operator*, make_rotation_{x, y, x}, make_translation() and make_perspective_projection().

After implementing these functions, we made a series of tests (using the Catch2 testing library) to ensure that each function was working correctly and outputting the right results. For all the tests, we used our own values to test the functions and manually calculated the results that should be outputted / known good values by hand and with online calculator tools. The tests would ultimately pass if the outputted values from the functions used in the tests matched our calculated values used in the test assertions.

In the test case for the Mat44f operator*, we defined two 4x4 matrices containing various differing values in each position and then multiplied them together. The expected resulting matrix was calculated using an online calculator tool and we then compared the value of each position in the resulting matrix with the expected value for that position (from our calculation). This was done to ensure that this function produced the right output and that matrices can be correctly multiplied in our program e.g. when performing both translations and rotations on an object.

For the Vec4f operator*, we reused one of the matrices from the previous test and also defined a new 4-vector and multiplied these together, again testing the output of this operation with values calculated using an online calculator. Like the previous test, this test was done to ensure any matrix-vector multiplications in our program were handled correctly and produced the right output.

We then tested the x, y and z rotation functions. For each of these functions we tested four things: rotating the matrix by nothing (should produce the same identity matrix used in for rotation tests), rotating the matrix by 90 degrees, rotating it by -90 degrees and rotating it by 180 degrees to test a range of rotation motion for each axis of rotation. This resulted in 12 tests in total. This was done in order to ensure that the resulting matrices output the correct values so that we could rotate objects in our program (i.e. the spaceship) correctly, without any issues.

For make_translation(), we tested that the matrix could be translated correctly for when the x coordinate was translated, the y coordinate was translated and the z coordinate was translated, as well as testing that the resulting matrix was correct when all three coordinates (x,y and z) were translated. This was done to ensure that any translations in our program occurred correctly i.e to moving the camera or translating the objects in the world.

Finally, we tested make_perspective_projection() by feeding the function a standard window size of 1280x720, a Field Of View (FOV) of 60 degrees, a near plane of 0.1 and a far plane of 100. We then tested that the resulting matrix for this operation matched our calculated expected values. This was done to ensure that our projections are correct so the program can function correctly and display the world appropriately, even when the window is resized.

1.2 3D Renderer Basics

We then began implementing the 3D renderer basics to render and display the world mesh for Parlahti.

We added global GL setup settings to main.cpp and successfully loaded in the Wavefront OBJ file for Parlahti in our program. Afterwards we implemented a first-person style 3D camera so that we could freely explore and examine this world mesh. The camera could be controlled using the WASD + EQ keys for a range of movement, as well as using shift to speed up and ctrl to slow down.

Furthermore, we also added a simplified directional light model with the light direction (0,1, -1). Evidence of all of this can be seen in Figures 1-4 below.

For our test computer, we had the following values for GL_RENDERER, GL_VENDOR and GL_VERSION:

- GL_RENDERER: Mesa Intel(R) UHD Graphics 630 (CFL GT2)
- GL_VENDOR: Intel
- GL_VERSION: 4.6 (Core Profile) Mesa 22.0.5

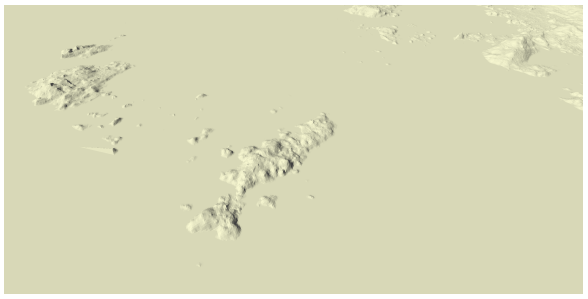

Figure 1: View of smaller islands in the world mesh

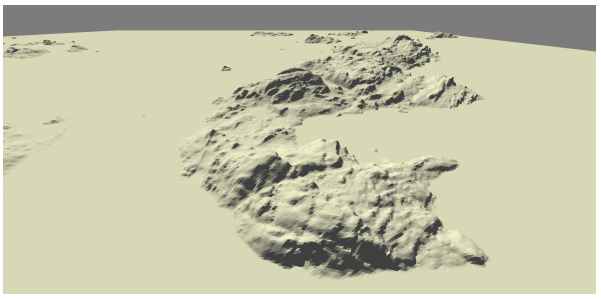

Figure 2: View of larger islands in the world mesh

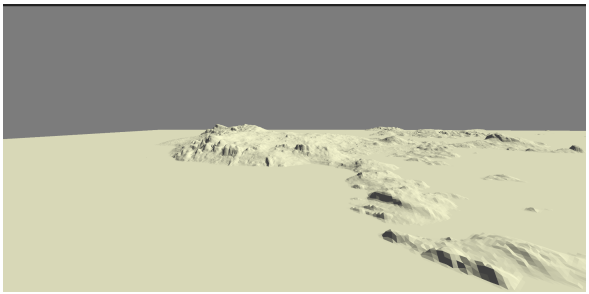

Figure 3: View of the world mesh

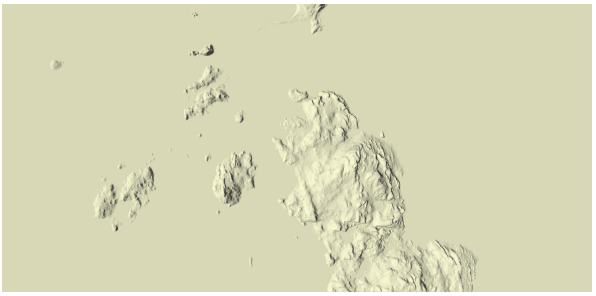

Figure 4: Top-down view of islands in world mesh

1.3 Texturing

We then continued by updating our renderer to draw the world mesh with a texture (orthophotos of Parlahti). We also combined the texture with the previous simple lighting implementation. Evidence of this is shown in Figures 5-8 below.

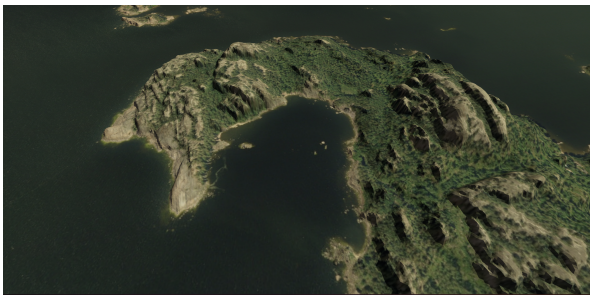

Figure 5: Larger island with texture mapping

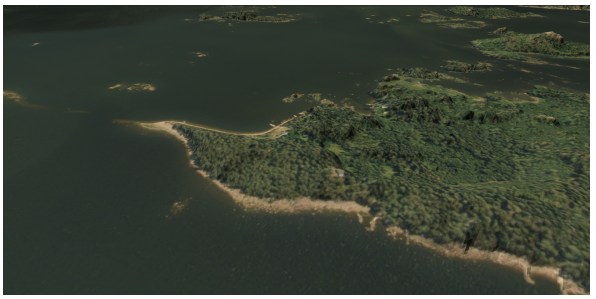

Figure 6: Another view of the world with textures

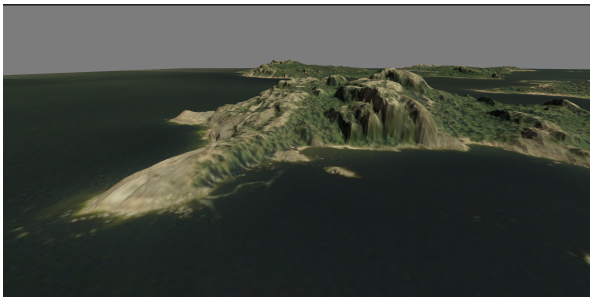

Figure 7: Island with texture mapping

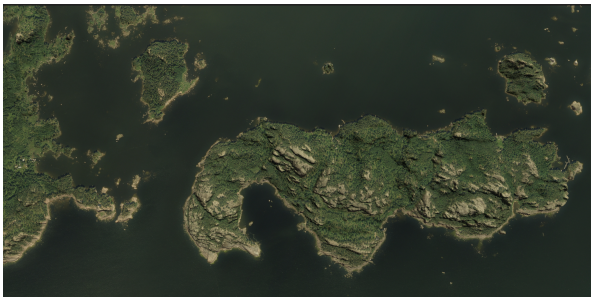

Figure 8: Top-down view of textured world

1.4 Simple Instancing

To further add to the program, we implemented simple instancing. We loaded in a second Wavefront OBJ file for the launch pad and rendered it twice (not copying the data) to make two launch pads in the world. The models were rendered with per-material colours. We then placed them both in the sea, in contact with the water (not submerged) and away from the origin. Evidence of this can be seen in Figures 9 and 10 below.

The coordinates of the launch pads are as follows:

- First Launch Pad: x: -24.5, y: -0.97, z: -54.0
- Second Launch Pad: x: -5.7, y: -0.97, z: -2.0

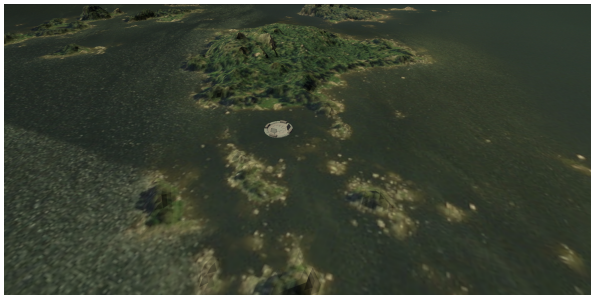

Figure 9: First Launch Pad placement

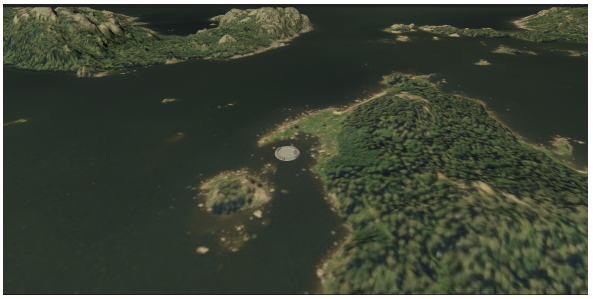

Figure 10: Second Launch pad placement

1.5 Custom Model

We also programmatically made a custom space vehicle model for use in the program. We designed it as a complex object composed of 10 total shape objects, made up of three different basic shapes (cube, cylinder and cone). All the shapes are connected to each other and all transformations (scaling, rotation, translation) were used in making the model. Also, we calculated the normals for all the shapes used in the spaceship so that it was lit appropriately. Evidence of this can be seen in Figures 11-13 below. Furthermore, our design was loosely inspired by NASA's Artemis SLS Rocket.

We then placed our space vehicle on the second launchpad at coordinates: x: -5.7, y: -0.97, z: -2.0, which can also be seen in the figures below.

Figure 11: Lit side of custom model

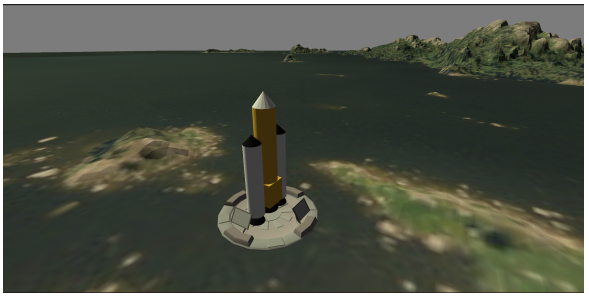

Figure 12: Shadowed side of custom model

Figure 13: Far view of custom model on the second launchpad

1.6 Local Light Sources

We then implemented the full Blinn-Phong shading model for point lights, with the standard distance attenuation. We added three point lights to the vehicle, each with a different colour: White at the bottom, red at one side and blue at the other side. This can be seen in Figures 14 and 15 below.

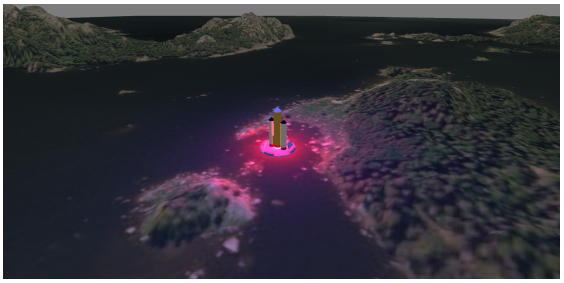

Figure 14: Ship and launch pad with lighting

Figure 15: Lit ship in air, with lighting also on ground

1.7 Animation

Furthermore, we added an animation where our space vehicle flies away, travelling along a curved path and slowly accelerating from a standstill. Our vehicle also rotates to face the direction of its movement and the lights attached from Section 1.6 follow the space vehicle. The animation started upon pressing the F key and would reset upon pressing the R key.

Evidence of all this can be seen in Figures 16 and 17 below.

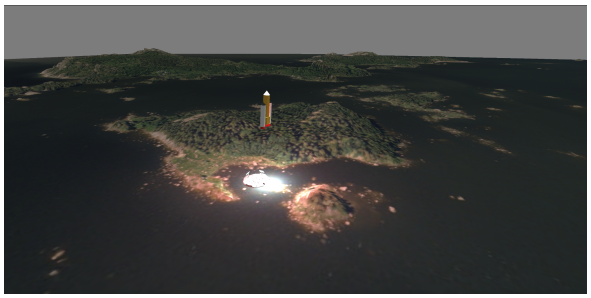

Figure 16: Vehicle beginning animation and 'taking off' (with attached lights shown)

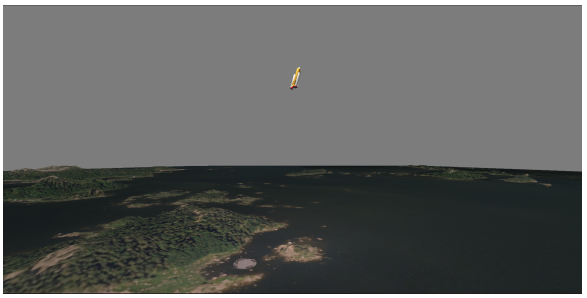

Figure 17: Vehicle moved away from launch pad and rotated towards direction of movement

1.8 Tracking Cameras

We implemented a feature where the user can switch between two camera modes: one that looks at the vehicle from a fixed distance and follows it in its flight and one that is fixed on the ground and looks at the spaceship as it flies away. Pressing the C key will cycle the user through these different camera modes from free camera to the fixed-distance camera to the ground camera and then back to the free camera again. These two additional camera modes can be seen in Figures 18 and 19 below.

Figure 18: Screenshot of fixed-distance camera mode

Figure 19: Screenshot of ground camera mode

1.9 Split Screen

We continued by implementing split screen functionality, however, we were unable to implement the full functionality for this task. Our current implementation allows for the screen to be split vertically (50% - 50%) and displays two screens. We did this by creating one viewport that draws all the objects to it when the split screen is toggled off and two viewports when the split screen is toggled on with the V key. When toggled, the two viewports are made with half of the width of the window and all of the objects are drawn twice, once to each viewport. Toggling camera modes for specific views is not implemented here, however pressing the C key will still change the camera modes (from Section 1.8), but will change the camera for both viewports.

Figure 20: Split-screen mode with camera tracking

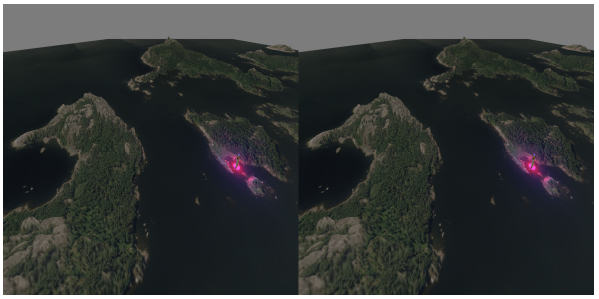

Figure 21: Split-screen mode with free-roam camera

1.10 Particles

Ultimately, we were unable to implement particles in full due to time constraints. An attempt was made to implement them, however the code remained mostly unused. We began by creating new files to handle particle and particle emitter functions. We defined a struct to handle particle attributes, which contained each particle's: position, velocity, colour and life (time for the particle to be displayed in the world).

We then created additional .frag and .vert shader files for the particles. After achieving a rough implementation of this and the particle functions, we started running out of time and ultimately decided to leave this task undone and the files / code created for it were not included in the final submission. We then spent our remaining time focusing on other tasks.

The plan for our implementation of particles in our program was as follows: complete function to render particles in the world and translate their positions relative the space vehicle, create a function to update the relevant particle attributes (e.g. velocity and age) and then define a statement to 'respawn' any 'dead' particles from the emitter to have a constant stream of particles at a lower resource cost (more efficient approach).

Theoretically, a limitation of this particle system is that it could be slightly resource intensive and potentially lower the frame rate (increase in time to draw each frame) once it has been implemented, which would be more noticeable on lower spec machines.

However, in our planned implementation, with the 'killing' and reusing of the particles, we have a maximum amount of particles that could be rendered on screen at a given time. This would make our particle system quite efficient and should result in a better performance when compared with other particle systems.

1.11 Simple 2D UI Elements

We were also unable to implement this section due to the aforementioned time constraints. However, with enough time we would have attempted to use Fontstash to render text and add an altitude tracker in the top left corner of the window, using the DroidSansMonoDotted.ttf font. As Fontstash is commonly used with legacy OpenGL and we are using modern OpenGL, we would need to implement our own version of functions found in the glfontstash.h file (which we don't have in this project), such as glfonsCreate() and glfonsRGBA(). We would then track the rocket's y-position (for altitude) and display that in the text UI; updating it for each frame to track the rocket's altitude.

For the launch and reset buttons in the bottom centre of the screen, we would have used an orthographic projection matrix, with the view and model matrices both being set as identity matrices, in order to have a 2D screen projection in which we could add UI elements like the buttons.

We could then define two objects / squares in that projection space and use glOrtho() to map the objects to our orthographic projection space (/ the 2D UI). Afterwards, we would use the coordinates of these two objects to create functions to change the appearance of these buttons upon hovering, as well as using glfw_callback_key() to change the appearance of the button on mouse click. To add text to these buttons we could again use Fontstash to render text over these buttons.

To add another UI element, we would repeat the steps noted in the paragraph above. We could then create another object and call glOrtho() to render it in our 2D projection space. We could then freely manipulate this object to add the desired UI element for our program.

1.12 Measuring Performance

Frame No.	Time Taken (ms)	
First Frame	Section 1.2 Rendering Time:	2.185456ms
	Section 1.4 Rendering Time:	0.000000ms
	Section 1.5 Rendering Time:	0.000000ms
	Frame Full Rendering Time:	0.000000ms
	Frame To Frame Time:	11.000000ms
	Submit Rendering Commands Time:	3.000000ms
Second Frame	Section 1.2 Rendering Time:	2.185456ms
	Section 1.4 Rendering Time:	0.000000ms
	Section 1.5 Rendering Time:	0.000083ms
	Frame Full Rendering Time:	0.000083ms
	Frame To Frame Time:	12.000000ms
	Submit Rendering Commands Time:	5.000000ms
Third Frame	Section 1.2 Rendering Time:	2.185456ms
	Section 1.4 Rendering Time:	0.000083ms
	Section 1.5 Rendering Time:	0.000083ms
	Frame Full Rendering Time:	0.000083ms
	Frame To Frame Time:	21.000000ms
	Submit Rendering Commands Time:	0.000000ms

Figure 22: Table of results for measuring the performance of our renderer

We then measured the performance of our renderer. We did this using two methods: OpenGL query objects and std::chrono::high_resolution_clock. To measure the full rendering time of each frame and the time to render Sections 1.2, 1.4 and 1.5, we created 4 GLuint variables and assigned a generated query for each of them respectively using glGenQueries. I then created 4 timestamp variables for these 4 queries. In the main rendering loop (while loop), I used a series of glBeginQuery() and glEndQuery() functions with GL_TIME_ELAPSED to track the time between these checkpoints / the render functions we want to measure. Afterwards we used glGetQueryObjectui64v() to get the result time of the query and save it to the timestamp variable. This variable was then used in a printf() statement to display the render time for that particular section in that frame.

Additionally, we used two std::chrono::high_resolution_clock::now() functions, enclosing the code to submit rendering commands. These values were subtracted from each other, the result was converted to milliseconds and this result was printed to the terminal to show the time taken to submit the rendering commands. Furthermore, we use this function again along with a variable declared outside the main rendering loop. This was done so that we could measure the frame-to-frame time (as we would need to track the time between iterations of the rendering loop). This time was calculated as the time needed to clear the first frame and re-render / draw the second frame. Like before, the result was calculated, converted to milliseconds and then printed to the terminal for visual verification.

From the table of results above (Figure 22), we can measure the performance of our renderer. As we can see from the table, the time taken to render Section 1.2 (the world mesh) stayed consistent throughout each frame. This is what we would expect as the world mesh never changes and is always visible on screen. Even with movement, this value does not change, which again can be explained by the fact that the world mesh remains the same and is never changed within our program, so it is always rendered the same.

For rendering Sections 1.4 and 1.5 (the two launch pads and the space vehicle respectively), the times taken varied between 0.000000ms and 0.000083ms, which is what we would expect of our renderer. It was also found that these results vary as we move and look around, which is also expected. The low value of 0.000083ms can be explained by the low computation cost needed to render the two launchpads and the space vehicle, since they are smaller, simpler objects with a lower number of vertices to account for. This results in a faster render time. Furthermore, we noticed that the 0.000000ms value appears when we are not looking at either of the launch pads or space vehicle, which makes sense as the launch pads and spaceships are not in the program's field of view and thus do not need to be drawn or rendered, which would result in this zero value. The outlier in the second frame where Section 1.4 renders with 0.000000ms and Section 1.5 renders in 0.000083ms can be explained due to us viewing the spaceship in the air (mid-animation) but not having the launch pad in the field of view at that time.

For rendering the full frame, we noticed that the times taken varied between 0.000000ms and 0.000083ms. It was 0.000000ms when Section 1.4 and Section 1.5's rendering times were also 0.000000ms and conversely, when either Section 1.4 or Section 1.5's rendering times were 0.000083ms, the frame render time was also that value, even when both 1.4 and 1.5 were being rendered (as they were in the field of view). This is mostly what we expect, as when the objects are being rendered, then the time to render the frame would at the very least match an object that is being rendered, however, we would hope that the total frame render time would be the sum of the times taken to render all objects in that frame. So we would probably like to see a value of ~2.185622ms here in the third frame.

The frame-to-frame time varied considerably across frames as we can see in the table, with a value of 11ms in the first frame, 12ms in the second and 21ms in the third. These results changed as we moved around our program and looked at different objects. These results are expected, as the time taken to clear and redraw the frames would without a doubt change as more variables are introduced such as animations, objects on screen etc. This would result in an increased render time and an increased time to clear the frame, which would ultimately increase the tracked frame-to-frame time.

Finally, we can see that the time taken to submit the rendering commands also varied like the frame-to-frame time before, with 3ms in the first frame, 5ms in the second frame and 0ms in the third frame. The results seen here do not line up with what we expected. We expected that as the time taken to render the objects and frame increased, then the time taken to submit the rendering commands would also increase due to the increased amount of processes to run and return. However, in the results we analysed, the times taken to submit these rendering commands seem almost random.

It should be noted that while the shown results were for one run in particular, the tests were run multiple times to verify our findings and the results were reproducible in different runs.

References

Anon. 2015. *LearnOpenGL - camera*. [Online]. [Accessed 8 December 2023]. Available from: https://learnopengl.com/Getting-started/Camera

Anon. 2015. *LearnOpenGL - basic lighting*. [Online]. [Accessed 9 December 2023]. Available from: https://learnopengl.com/Lighting/Basic-Lighting

Anon. 2015. *LearnOpenGL - advanced lighting*. [Online]. [Accessed 10 December 2023]. Available from: https://learnopengl.com/Advanced-Lighting/Advanced-Lighting

Appendix

Tasks	Contributions
1.1 Matrix / Vector Functions	Functions implemented - Kieran Test cases implemented - Aodhan
1.2 3D Renderer Basics	Mesh rendered and displayed - Kieran Lighting - Kieran Camera functionality - Jibran
1.3 Texturing	Texturing - Kieran
1.4 Simple Instancing	Simple Instancing - Kieran
1.5 Custom Model	Normals calculation - Kieran & Aodhan Model - Aodhan
1.6 Local Light Sources	Local lighting - Jibran
1.7 Animation	Animations - Kieran & Aodhan
1.8 Tracking Cameras	Tracking Cameras - Kieran
1.9 Split Screen	Split Screen - Kieran
1.10 Particles	Particles - Jibran
1.11 Simple 2D UI Elements	Simple 2D UI - Aodhan & Jibran
1.12 Measuring Performance	Measuring Performance - Aodhan
Report	Report - Aodhan