1 Introduction

In this report I will discuss my implementation of 4 requirements of the WebGL coursework, the motivation behind my solutions, the issues encountered when implementing them, potential future improvements, and the mathematics involved in developing each requirement. For each requirement I will attempt to analyse as critically as possible my approaches and areas of potential improvement, as well as solutions which I believe to be particularly efficient and or simple.

2 Requirement 2 - Draw co-ordinate system axes

For this requirement I needed to display the three co-ordinate axes using different coloured lines representing the x, y, and z axes respectively. The task at first appeared simple, drawing three straight lines along the three axes using the `THREE.line(geometry, material)` method in ThreeJs. Drawing the y axis line in this way caused no issues however when drawing the red and blue lines along the x and z axes I encountered an issue known as *Z-Fighting*. The portion of the lines nearest to the origin were 'fighting' with the mesh grid over priority in being drawn to the screen first, causing an unclear blur of the white line from the mesh grid and the coloured line from the axes to be displayed.

Z fighting occurs when two (or more) primitives have very similar distances to the camera in the scene [1]. A problem then arises as the two primitives have nearly identical values in the *Z-Buffer*. The Z-Buffer keeps track of the depth of each primitive to be rendered to the screen in order to calculate which objects need to be displayed over top of one another. When two primitives have such similar values in the Z buffer it is almost random which of the primitives is drawn in a given pixel as the Z buffer struggles to distinguish which one is closer to the camera. As a result of this the white artifacts seen in **Figure 1** are formed on the x and z axes (note that the y axis is unaffected as the mesh grid is not drawn in the y direction).

There are a number of ways of fixing this Z fighting issue including adding a slight offset to the position of the mesh grid however the particular approach I used was to add a separate scene specifically for the axis lines which is then drawn over top the scene containing the mesh grid and all of the other objects. This is a simple solution to the problem and this method of fixing the Z fighting issue allows me to add any number of items to the scene containing the mesh grid with the confidence that the axis lines will be drawn over top of all of them regardless. A criticism of this approach however is that it is more computationally expensive to add another scene to the environment as this new scene needs its own lighting, rendering, etc. A better implementation may perhaps be to specify that the mesh grid has lower Z-Buffer priority than every other object in the scene alleviating Z-Fighting issues as the mesh grid will always have a lower priority over every other object.

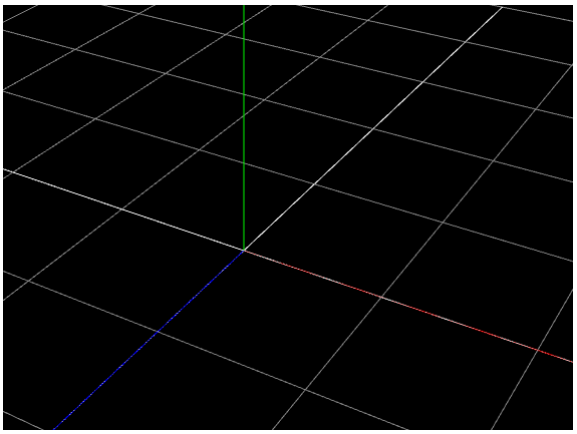
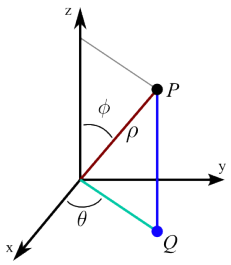


Figure 1: Z fighting issues along the z and x axes

3 Requirement 6 - Longitudinal camera orbit

In order to complete this requirement I needed to orbit the camera about the cube at a fixed look at point in both latitudinal and longitudinal directions. Whilst discussing my solution I will focus solely on the longitudinal orbit as this caused more issues to implement and as the latitudinal orbit is a simplified version of the longitudinal solution. My initial idea was to model the camera as a point on a sphere with its centre at the look at point, the centre of the cube. To move the camera I would then use the known equations of a sphere to change the camera's position on the sphere in order to orbit it about the cube longitudinally. The mathematics behind this is known as *spherical polar co-ordinates* which can be used to describe a point on a sphere using the radius of the sphere, an angle ϕ and an angle θ. For point **P** if we ignore the z component and project **P** onto the xy plane and call this **Q** then θ is the angle **Q** makes with the positive x axis. ϕ is the angle between **P** and the positive z axis [2]. Using trigonometry we can then calculate the polar to Cartesian conversions for 3D polar co-ordinates:



$$x = \rho \sin\phi \cos\theta$$
$$y = \rho \sin\phi \sin\theta$$
$$z = \rho \cos\phi.$$

Figure 2: Visual representation of polar co-ordinates [3] and the corresponding conversion to Cartesian co-ordinates

I soon learned that the axes in WebGL were not oriented in the same way as the axes in standard spherical co-ordinates. The y axis in WebGL corresponds to the z axis in the above equations, and vice versa for the z axis corresponding to the y. I then implemented the equations and incremented the value of ϕ in order to orbit the camera over top the cube. This required me to initialise ϕ and θ based off of the camera's initial position (this code is not included below). The main limitation of my approach is that having moved the camera I then update its look-at vectors using the method `camera.lookAt(x, y, z)`. The problem with using this method is that it causes gimbal lock at the top-most (highest y) and bottom-most (lowest y) point of the orbit. This presents as the camera moving up over-top the cube, turning 180 degrees to be facing the opposite direction, and then moving down the back face of the cube. The camera is thus still orbiting the cube longitudinally however due to this gimbal lock effect the animation appears somewhat unpolished. For a better implementation I may update the camera's look-at vectors myself to avoid this occurring rather than use the built in method.

```
function longitudinalCameraOrbit(){
        const speed = Math.PI/200;
        const radius = cameraPos.distanceTo(new THREE.Vector3(0, 0, 0));

        const newX = radius * Math.sin(phi) * Math.cos(theta);
        const newZ = radius * Math.sin(phi) * Math.sin(theta);
        const newY = radius * Math.cos(phi);

        camera.position.set(newX, newY, newZ);
        phi += speed
}
```

4 Requirement 8 - Load the bunny model

To meet this requirement, I needed to load in the Stanford bunny from its `.obj` file and display it neatly inside the cube. To do this I used the `OBJLoader` class to load in the file. Files in the format of `.obj` store the object by first listing all of its vertices and then listing all of its faces which are simply the indices of 3 vertices from which a surface is formed. I encountered issues with loading the bunny due to the fact that in WebGL using this `OBJLoader` class the bunny is loaded *asynchronously*. Since loading in an object from a file could potentially take quite some time the `onLoad()` function of the OBJLoader is executed asynchronously meaning that the code inside the `onLoad()` function may be executed out of sequence. Essentially JavaScript will 'juggle' between operations when executing asynchronous functions (whilst remaining single-threaded). This is a simplified explanation of what happens between the call stack and the execution stack when running an asynchronous function in JavaScript, the program will periodically *call-back* to the call stack [4] when executing these large, potentially time consuming operations. My explanation of asynchronous functions here is a simplification in order to demonstrate the issues I had.

I will touch upon how asynchronous operations can be handled in the code in my discussion of improvements to my implementation.

My issues became apparent when I attempted to refer to the bunny object in a variable in the `onLoad()` function. The problem with doing this is that there is no guarantee that the bunny has been fully loaded when referencing it in a variable hence often times I would attempt to manipulate the bunny variable whose contents were null. The solution to this is to execute all required manipulations (such as scaling it to fit inside the cube) of my bunny within the `onLoad()` function. This ensures that once the bunny is loaded it will have been correctly handled as I wished it to be.

```
function loadBunny(){
        const loader = new THREE.OBJLoader();
        loader.load('bunny-5000.obj',
        function (object) {
                //All relevant bunny manipulations...
        });
}
```

As mentioned previously there are ways of managing the asynchronous load function, in fact this can be done in such a way as to allow me to reference the bunny object inside a variable by using callbacks and *promises*. This would be a definite improvement upon my solution which I would implement given more time as it allows for more flexibility than my current implementation. Having said that the code which I have written still ensures the bunny will be loaded and manipulated correctly and in this way it is perfectly suitable as a solution to the requirement as I have overcome my initial issues with the asynchronous nature of the operation.

5 Requirement 10 - Do something cool

For the 'do something cool' requirement I decided to experiment with shaders in WebGL using a well known and widely used algorithm in computer graphics called *Perlin noise*. In my implementation I have used 3D Perlin noise however for the purposes of explanation I will be examining the mathematics behind 2D Perlin noise. 3D noise works in the exact same way except instead of using squares on a grid with 4 vertices each the noise is calculated using cubes with 8 vertices in 3 dimensions. I will initially explain how the algorithm works in 2D and then show my own application using shaders and a sphere object in WebGL.

Firstly define our 2D Perlin noise function `noise = f(x, y)` as returning a *scalar* value where `(x, y)` is any point in 2D. Now consider a 2D grid with integer values (x, y) marked as points on the grid. For every grid point a gradient vector **G** is created (typically of uniform length 1) pointing in a random direction from said grid point. Note that this is done *pseudo-randomly* i.e. given a specific seed the same random gradients will be generated every time. Consider a pixel **P** that we need to compute noise for bounded by 4 points in the grid. For each of these neighbouring grid points **Q** we define a distance vector **D**

that is the distance between P and Q. We then take the dot product of the gradient vector G and the distance vector D for each point Q and thus generate 4 values contributing to the final noise value at P. The final step of the algorithm is interpolation between these 4 (8 in 3D) values. The choice of interpolation algorithm varies, in Ken Perlin's original implementation a modification of the *Smootherstep* [5] algorithm was used to interpolate 8 points (as the original implementation was in 3D). Having interpolated the 4 values we now have a scalar noise value generated for a point `(x, y)` in 2D.

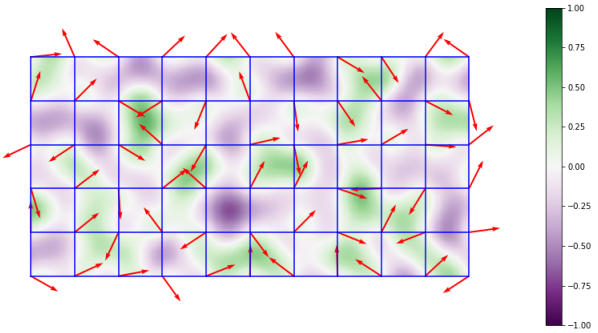


Figure 3: A 2D Perlin Noise map after interpolation. Red arrows represent gradient vectors

In order to allow for quick computation, I implemented Perlin Noise as a vertex shader in WebGL. Shaders work directly with the GPU and are applied to every vertex of a given object in WebGL in parallel leading to efficient and fast computation. The caveat of this however is that shaders are written in their own special shader language which has its own unique set of challenges to overcome during implementation. Fortunately I found a 3D Perlin Noise shader online [6] which had all of the necessary calculations included for implementing the noise algorithm. I adapted this shader myself to work with my own project and wrote my own fragment shader (that is a shader which manipulates the colour of the object it's applied to) which produces a smooth gradient mixture of 2 colours [7]. Using the `THREE.SphereGeometry(...)` method I created a sphere with 32 width segments and 64 height segments and rendered it to the screen. With use of a global `noiseFactor` variable for controlling the 'degree of randomness' of the generated noise i.e. how wildly the newly generated vertices with noise applied will vary from their original positions, I applied 3D Perlin noise to each vertex (that is a point corresponding to width and height segments) of the sphere. By stepping the `noiseFactor` up by a small increment every animation frame and then recomputing the noise applied to the vertices the sphere appears animated and moving, with peaks on the sphere becoming increasingly more wild and of greater magnitude, until eventually after enough frames of animation the sphere appears to be morphing out of control with spiky peaks and wild movement.

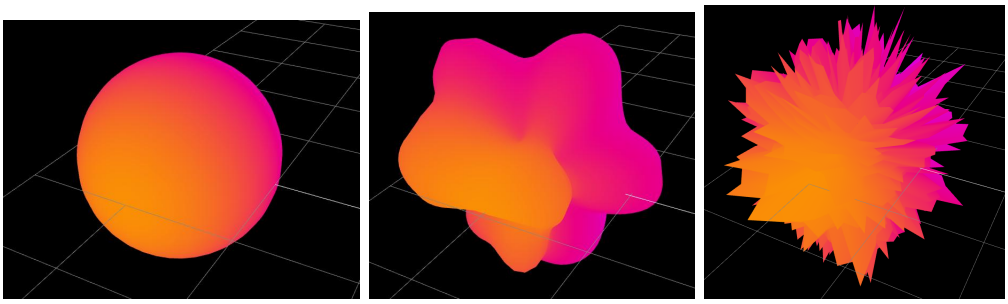


Figure 4: Left: The original sphere, Centre: The sphere with a moderate degree of noise applied, Right: The sphere with a high degree of noise applied

The use of a vertex shader in applying noise makes for efficient computation of the vertices' new values, and the animation runs smoothly without strain on the CPU (as the GPU is being used for calculations). The limitations of my implementation are in that it would be useful to allow the user to control the degree of noise applied via a slider rather than simply switching the animation on or off. This could be achieved by use of a GUI which I would implement had I had more time on the project. Another feature that I would add given more time is perhaps to allow the user to save certain states by preserving the `noiseFactor` of a specific stage of animation that they would like to view again and allowing them to load in any state given a `noiseFactor` and gradient seed.

References

[1] Mert Akça (2020), *Depth Buffer and Z-fighting*, DevGenuis, https://blog.devgenius.io/computer-graphics-depth-buffer-test-5c29807cf475

[2] Weisstein, Eric W, *Spherical Coordinates*, MathWorld–A Wolfram Web Resource, https://mathworld.wolfram.com/SphericalCoordinates.html

[3] Nykamp DQ, *Spherical Coordinates*, Math Insight, http://mathinsight.org/spherical_coordinates

[4] Tapas Adhikary (2021), *Synchronous vs Asynchronous JavaScript – Call Stack, Promises, and More*, freeCodeCamp, https://www.freecodecamp.org/news/synchronous-vs-asynchronous-in-javascript/

[5] Ken Perlin (1997), *Coherent noise function over 1, 2, or 3 dimensions*, New York University, https://cs.nyu.edu/perlin/doc/oscar

[6] Stefan Gustavson, *Classic 3D Perlin Noise*, Noise algorithms, https://gist.github.com/patriciogonzalezvivo/670c22f3966e662d2f83

[7] Patricio Gonzalez Vivo, Jen Lowe, *Colors - The Book of Shaders*, The Book of Shaders, https://thebookofshaders.com/06/