CM20219 – Fundamentals of Visual Computing: VIEWING AND ANALYSING 3D MODELS USING WEBGL

Candidate Number: 10025, *CM20219, University of Bath*

Abstract The documentation of brief series of exercises demonstrating the capabilities of the industry standard graphics programming API, WebGL. For the purposes of this coursework, the *Three.js* library wrapper is used to enable the use 3D animations using the JavaScript language. As well as this, two other libraries are utilized: *OBJloader*, to enable the import of external models; and *dat.GUI* which allows for basic menu-style GUIs to be implemented. What follows is an overview of object creation and manipulation; camera control; lighting and shade; texture mapping; GUI creation; and rendering; all using JavaScript and the afore mentioned libraries.

I. INITIALISING A WEBGL SCENE

BEFORE it is possible to begin implementing *Three.js* features, there are a few prerequisites which will nearly always be required. First, as the JavaScript will be embedded in a HTML file, it is good practice to define the files metadata. Setting the file's character set, title and style is not essential; but the metadata can also contain references to JavaScript source libraries, so they have all been linked here before the <*body*> of the HTML file.

```
<html>
    <head>
        <meta charset=utf-8>
        <title>OpenGL Assignment</title>
        <style>
            body { margin: 0; }
            canvas { width: 100%; height: 100% }
        </style>
        <script src="js/three.js"></script>
        <script src="js/dat.GUI.js"></script>
        <script src="js/OBJloader.js"></script>
    </head>
    <body>
```

Code Extract 1. The code that precedes the body of the HTML file showing the linking of the *Three.js*, *OBJloader.js* and *dat.GUI.js* JavaScript files.

Once all the required libraries have been linked, it is possible to create a WebGL scene using *Three.js*, at this time the objects *camera* and *renderer* can be created and initialised as well as the *render* function to be used for animating objects later.

The *camera* is initialised with arguments that define its viewing frustum [1], the region of the rendered space that appears on screen. The *camera* position is then set by `camera.position.set( 5, 5, 5 )` which adjusts the x, y and z coordinates respectively, preventing objects loaded in at the scene origin from clipping around the camera.

The *renderer* is initialised using the code shown below, notice that show mapping is enabled and the type is set to *PCFSoftShadowMap* to provide better shadows when implemented later.

```
// Initialise Renderer
  var renderer = new THREE.WebGLRenderer();
  renderer.setSize( window.innerWidth, window.innerHeight );
  document.body.appendChild( renderer.domElement );
  renderer.shadowMapEnabled = true;
  renderer.shadowMapType = THREE.PCFSoftShadowMap;
```

Code Extract 2. Initialisation of *Three.js* WebGL renderer with shadow mapping enabled.

II. SPECIFICATION REQUIREMENTS

A. The Cube

This section covers all cube related aspects of the specification as well as requirement number 2. Draw coordinate system axes. The remaining requirement numbers in this section are: 1. Draw a simple cube; 3. Rotate the cube; 4. Different render modes; and 7. Texture mapping.

Creating and rendering a basic 3D object in *Three.js* requires a minimum of 3 variables, these are: the geometry that defines the shape; the material that will wrap the object; and the mesh that combines these two elements. The code required to generate and add a cube to the scene is shown in *Code Extract 3.*, note that a *TextureLoader* is used to import a *.jpg* file that is then used as the material map. The material type is also that of *MeshLambertMaterial* as this allows the object to

receive and cast shadows. When an object is added to a scene with no other referencing (*Code Extract 3*), it is always added at the scene's origin; whether or not it visually appears to be located at the origin depends on the center point of the model or object itself.

```
var geo_cube = new THREE.BoxGeometry( 2, 2, 2 );
var tex_cube = new
THREE.TextureLoader().load( 'textures/crate1.jpg' );
var mat_cube = new
THREE.MeshLambertMaterial( { map:tex_cube } );
var cube = new THREE.Mesh( geo_cube, mat_cube );
scene.add( cube );
```

Code Extract 3. How to generate, texture and add a cube to a scene using *Three.js.*
Note: This is not a direct code extract, it has been pieced together for continuity purposes

Fig. 1 shows this cube rendered in the virtual environment with the *crate1.jpg* texture mapped on to it, this image also shows the scene axis with Red, Green and Blue corresponding to the X, Y and Z directions. **[REQ# 2]**

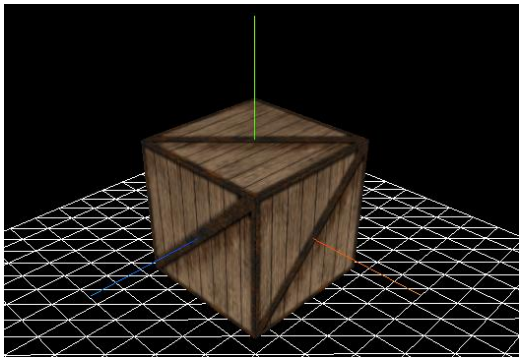

Fig. 1. Cube with dimensions of 2x2x2, centered at the scene origin (0, 0, 0) with a texture overlay; scene axes is displayed in RGB correlating to the XYZ directions.

It is possible to verify the dimensions of the cube by generating a bounding box from the cube object; the output from *Code Extract 4* can be seen in Fig. 2. As expected, all dimensions are equal to 2 which means, given that the object is centered at the scene origin, that opposing corner points can be found at (-1, -1, -1) and (1, 1, 1). **[REQ# 1]**

```
var bbox_cube = new THREE.Box3().setFromObject( cube );
alert("Dimension X: " + (bbox_cube.max.x - bbox_cube.min.x) +
      "\nDimension Y: " + (bbox_cube.max.y - bbox_cube.min.y) +
      "\nDimension Z: " + (bbox_cube.max.z - bbox_cube.min.z));
```

Code Extract 4. Generating bounding box to verify cube dimensions and displaying values via a JavaScript alert.

Fig. 2. Alert output from *Code Extract 4.* displaying the absolute dimensions of the cube.

Cube rotation is fairly easy to accomplish and is performed in the *render* function mentioned earlier in *I. Initialising a WebGL Scene*. For the *render* function to work, the browser must call *requestAnimationFrame(render)* from within the *render* function itself; this ensures successive frames are displayed onscreen before they are updated [2]. After the next frame has been requested, it is possible to update object parameters (in this instance the cube's X, Y and Z rotation) before the renderer repaints the scene. **[REQ# 3]**

```
var render = function() {
  requestAnimationFrame(render);

  cube.rotation.x += 0.01;
  cube.rotation.y += 0.01;
  cube.rotation.z += 0.01;

  renderer.render(scene, camera);
};
render();
```

Code Extract 5. How the cube is rotated within the scene and displayed by the browser.
Note: The call of *render()* at the bottom of the extract starts an infinite loop, this is due to the call of *render()* inside the function itself.

Rendering the cube in different modes essentially requires the generation of separate geometry using the parameters of the preexisting cube object. The face rendering shown in Fig. 1 is achieved by simply applying any solid texture to the cube as demonstrated in *Code Extract 3.* **[REQ# 4C]**

To render the 8 vertices of the cube, a separate piece of geometry is defined so it can be displayed at each of the vertices' locations; in this instance, a sphere with a radius of 0.1. A new material is also created, but this time is just given a solid colour so that it stands out in the scene. *Code Extract 6* shows the generation of the vertices by indexing the length of the *cube.vertices* array. For each vector this array contains, a new object is generated and added to a group; this makes manipulating all vertices simultaneously easer in the future.

```
var geo_vert = new THREE.SphereGeometry( 0.1, 50, 50 );
var mat_vert = new THREE.MeshLambertMaterial({color: 0xffff00});
var vertices = new THREE.Group();
for ( var i = 0; i < geo_cube.vertices.length; i ++ ) {
  var vertex = new THREE.Mesh( geo_vert, mat_vert );
  vertex.position.x = geo_cube.vertices[i].x;
  vertex.position.y = geo_cube.vertices[i].y;
  vertex.position.z = geo_cube.vertices[i].z;
  vertices.add( vertex );
}
```

Code Extract 6. Generating vertices by indexing cube object.
Note: This is not a direct code extract, it has been pieced together for continuity purposes

Fig. 3 shows the cube in vertex only render mode. **[REQ# 4A]**

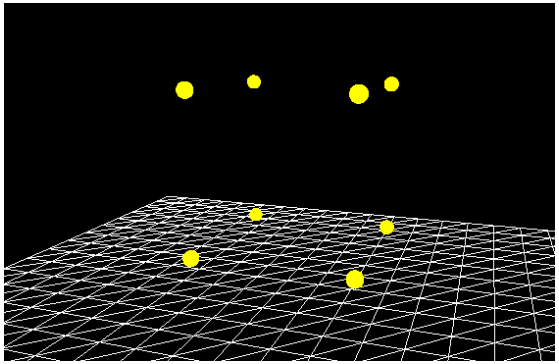

Fig. 3. Cube displayed in vertex only render mode.

The last render mode is edge only render, this can be achieved by simply generating edge geometry from the cube object and combining it with a texture using *THREE.LineSegments*; as shown in *Code Extract 7*. The in-browser render in show in Fig. 4. **[REQ# 4B]**

```
var cube_edge = new THREE.EdgesGeometry( geo_cube );
var mat_vert = new THREE.MeshBasicMaterial( {color: 0xffff00} );
var wire_cube = new THREE.LineSegments( cube_edge, mat_edge);
```

Code Extract 7. Generating edges from cube object parameters.
Note: This is not a direct code extract, it has been pieced together for continuity purposes

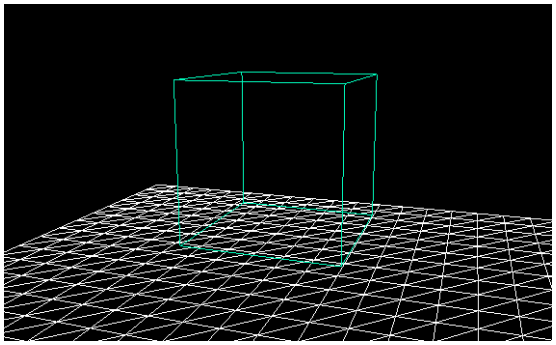

Fig. 4. Cube displayed in edge only render mode.

Retexturing requires prompting the object material to update once a new texture has been loaded into the material map. Fig. 5 shows a wood texture loaded onto the cube's current material map using `THREE.ImageUtils.loadTexture`. Once the new image has been loaded, it is updated for the next animation frame by using `cube.material.needsUpdate = true`. **[REQ# 7]**

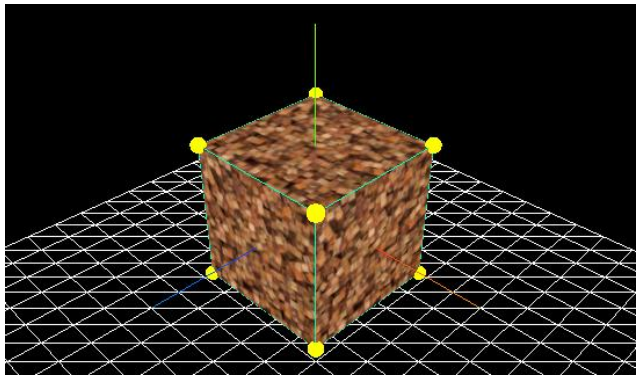

Fig. 5. Cube with a different texture loaded into the original material map.

B. The Camera

This section covers the camera functions as described in requirement numbers: 5. Translate the camera; and 6. Orbit the camera.

Translation is easily accomplished by manipulating the camera's coordinate vectors. This can be done a number of ways, for example *camea.position.set()* will set the camera's new position corresponding to the input arguments. Directly addressing the camera's parameters in another viable option i.e. *camera.position.x += 10* will translate that camera by +10 in the X direction.

However, in this particular application, a GUI is used to translate the camera as it provides a clear and easy way for the user to manipulate the camera's position as well as understand the camera's relative location in the render space. **[REQ# 5]**

```
var gui = new dat.GUI();

var cam = gui.addFolder('Camera');
cam.add(camera.position, 'x', -50, 50).listen();
cam.add(camera.position, 'y', -50, 50).listen();
cam.add(camera.position, 'z', -50, 50).listen();
cam.open();
```

Code Extract 8. GUI for camera translation controls
Note: This is not a direct code extract, it has been pieced together for continuity purposes

This allows the user to move the camera between ±50 about the origin of the scene. The function library *dat.GUI* provides a control slider by default to continuous variables and the camera position is updated as soon as the user moves the slider. More about GUI's can be found in *Section D* of this paper.

Orbiting around a point is quite a complex function as it requires a certain amount of knowledge about the orbit space in order to be successful. This particular solution consists of five different variables:

1. *p – the 3D Pythagorean distance to the orbit point from the camera*
2. *lon – the camera's longitude around the orbit point*
3. *lat – the camera's latitude around the orbit point*
4. *vec – a vector from the camera's location to the orbit point*
5. *point – a vector storing the orbit location*

Variables *p*, *lon* and *lat* are the spherical coordinates that allow for the repositioning of the camera within the 3D space; while *vec* and *point* enable orbits around a point other than the scene origin.

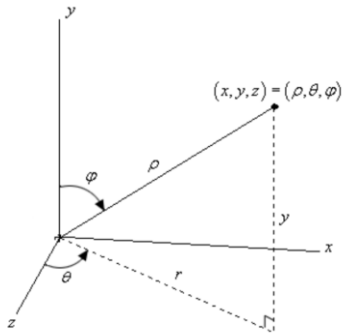

Fig. 6. Diagram showing the spherical coordinates systems [3]

The X, Y and Z positions of the camera can be defined by four equations using spherical coordinates:

$$\text{a)}\ z = \rho \sin\varphi \cos\theta \quad \text{b)}\ x = \rho \sin\varphi \sin\theta \quad \text{c)}\ y = \rho \cos\varphi$$

$$\text{d)}\ x^2 + y^2 + z^2 = \rho^2$$

$$\text{Where,}\quad \rho \geq 0 \quad \text{and} \quad 0 \leq \varphi \leq \pi$$

Equations 1. Spherical coordinate equations and constraints. [8]

There is also use for rearranged versions of the equations for Z and Y, in which θ and φ are made the subject. These equations are used to set the longitude and latitude from camera positon.

$$\text{a)}\ \theta = \cos^{-1}\frac{z}{\rho \sin\varphi} \quad \text{b)}\ \varphi = \cos^{-1}\frac{y}{\rho}$$

Equations 2. Latitude (θ) and Longitude (φ) equations.

The method by which an orbit is then achieved is as follows:

Initialisation

1. When orbit mode is enabled, update the orbit *point* using the orbit vector.
2. When orbit mode is disabled, update the orbit *vec* using the orbit point.
3. Calculate *p* using the camera position. (Equation 1d.)
4. Calculate *lon* using *p* and *camera.y*. (Equation 2b.)
5. Calculate *lat* using *lon*, *p* and *camera.z*. (Equation 2a.)

Render Function

6. Call orbit function if orbit is enabled.
7. Adjust *lon* and/or *lat* depending on orbit type.
8. Set *p* using the camera position. (Equation 1d.)
9. Set *camera.z* equal to Equation 1a. + *orbit.point.z.*
10. Set *camera.x* equal to Equation 1b. + *orbit.point.x.*
11. Set *camera.y* equal to Equation 1c. + *orbit.point.y*.
12. Set camera look at point to *orbit.point.*

The initialisation of the orbit sequence captures the current focal point of the camera and uses this as the new orbit point. The render function is called every time the browser requests a new animation frame so is effectively looped until orbit mode is disabled. **[REQ# 6]**

Note: The orbit function can be seen in *Code Extract 9*. This also shows some of the GUI options that help enable the orbit controls, these are explained in further detail in *Section D.*

```
function setOrbit() {
  if ( options.orbit.latitude == "Clockwise" ){
    options.orbit.lat -= 0.01;
  }
  else if ( options.orbit.latitude == "Counter Clockwise" ){
    options.orbit.lat += 0.01;
  }
  if ( options.orbit.longitude == "Ascending" ){
    options.orbit.lon -= 0.01;
  }
  else if ( options.orbit.longitude == "Descending" ){
    options.orbit.lon += 0.01;
  }

  var p = Math.sqrt(
          Math.pow(camera.position.x - options.orbit.point.x, 2)
        + Math.pow(camera.position.y - options.orbit.point.y, 2)
      + Math.pow(camera.position.z - options.orbit.point.z, 2));

  var z = (p * Math.sin(options.orbit.lon)
        * Math.cos(options.orbit.lat)) + options.orbit.point.z;
  var x = (p * Math.sin(options.orbit.lon)
        * Math.sin(options.orbit.lat)) + options.orbit.point.x;
  var y = (p * Math.cos(options.orbit.lon))
        + options.orbit.point.y;

  camera.position.set( x, y, z );
  if ( options.camera.focus_cube == true ) {
    camera.lookAt( cube.position );
  }
  else {
    camera.lookAt( options.orbit.point );
  }
};
```

Code Extract 9. Orbit function called from *render()* function when *options.orbit.enabled = true*.

C. *External Objects*

This section covers specification points relating to the importing of external objects and their manipulation; namely, specification points 8. Load a mesh model from .obj; and 9. Rotate the mesh and render it in different modes.

An external function library called *OBJloader.js* will be used to load the .obj into the environment. Prior to this a new material called *tex_blue* had been created which will be used to texture the object. All of the objects parameters must be set before it is added to the scene. First, a new instance of OBJloader is created, after which the object can be loaded with the *.load* function. The *.load* calls another function when the object is loaded, it is here that any object manipulations should be performed. **[REQ# 8.A]**

When the object is loaded into the environment it already has many children; in order to apply texture or shadow, the relevant parameters must be set for each child. To do this the structure must be traversed as show in *Code Extract 10*. **[REQ# 9.D]**

```
var loader = new THREE.OBJLoader();
loader.load('models/bunny-5000.obj', function ( obj ) {
  obj.traverse( function ( child ) {
    if ( child instanceof THREE.Mesh ) {
      child.material.map = tex_blue;
       child.castShadow = true;
    }
} );
```

Code Extract 10. Object-child traversal to set texture and shadow casting parameters.

Now the object has been properly mapped and textured, it is time to scale the model. A bounding box is created for the object in the same way the dimensions of the cube were checked in *Section A*. The scales factor is calculated by dividing the dimension of the cube by the largest dimension of the object bounding box; a matrix is applied to the object to align it with the origin before the scaling factor is applied to the model. **[REQ# 8.B]**

```
var bbox_bunny = new THREE.Box3().setFromObject( obj );
var scale = 2 / Math.max( bbox_bunny.max.x - bbox_bunny.min.x,
                          bbox_bunny.max.y - bbox_bunny.min.y,
                          bbox_bunny.max.z - bbox_bunny.min.z );
obj.applyMatrix(
  new THREE.Matrix4().makeTranslation( -0.5, 0, 0 )
);
obj.scale.set( scale, scale, scale );
```

Code Extract 11. Aligning and Scaling the object.

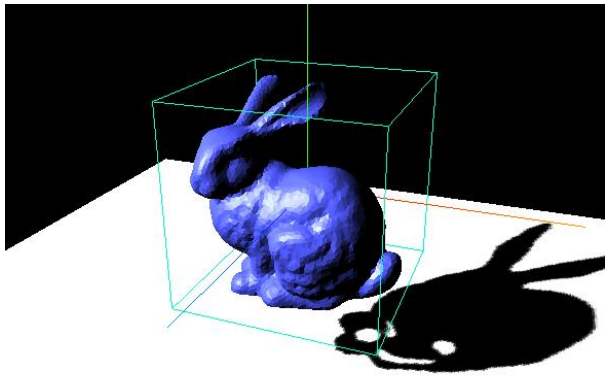

Fig. 7. Stanford Bunny loaded into scene with shadows and textures.

To get the imported object to rotate, it is added to another cube whose material

is hidden; this prevents the object from disappearing from the scene when the original cube is removed, but still allows full rotation control. **[REQ# 9A]**

```
var hidden_cube = new THREE.Mesh( geo_cube, mat_hidden );
hidden_cube.material.visible = false;
…
hidden_cube.add( obj );
…
hidden_cube.rotation.x += 0.01;
hidden_cube.rotation.y += 0.01;
hidden_cube.rotation.z += 0.01;
```

Code Extract 11. Method to rotate Stanford Bunny.

D. Extensions **[REQ# 10]**

As mentioned earlier, the extension to this task includes the addition of a graphical user interface and lighting to the scene. Parts of the lighting have been referenced throughout this document, however this hope to provide a more clarified explanation.

This example utilises two *pointlights*, positioned near the origin, to cast shadows within the scene. The two lights are grouped into an array and their parameters adjusted individually, even though many of these are identical.

```
var lights = [];
lights[ 0 ] = new THREE.PointLight( 0xffffff, 1, 0 );
lights[ 1 ] = new THREE.PointLight( 0xffffff, 1, 0 );
lights[0].castShadow = true;
lights[1].castShadow = true;
lights[ 0 ].position.set( -6, 6, 6 );
lights[ 1 ].position.set( -7, 7, 7 );
scene.add( lights[ 0 ] );
scene.add( lights[ 1 ] );
```

Code Extract 12. Initialising *pointlights.*

Pointlights emit light in all directions from their location and are initialised with their colour, intensity and distance (0 being infinite distance). [4] It is important to ensure that *.castShadow* is set to true, else the light wont be capable of producing shows on objects. Any object that also needs to cast a shadow can do so using *object.castShadow = true*; and objects that need to receive shadow (i.e. a ground plane) require *object.recieveShadow = true.*

Regarding render settings, as mentioned before, *render.shadowMapEnabled* must be *true* and *render.shadowMapType* must be set to *THREE.PCFSoftShadowMap.*

The second part of the extension includes the addition of a graphical user interface. This required the external *gui.DAT.js* library in conjunction with an options variable with stores various adjustable values as well as variables related to camera orbit.

A new GUI is created using *var gui = new dat.GUI*, from here folder can be created to group up related controls: *gui.addFolder().* It is important to ensure all controls *.listen()* as this will force them to update when their values are modified internally (not by the user). To increase functionality of a control, *.onChange(function (value)){}* is used to call a function when a control is interacted with. The value of the control can be passed to the function where it can be parsed using a suitable method; *if* and *switch* statements often do this very well and have been used here to enable orbit and change render mode respectively.

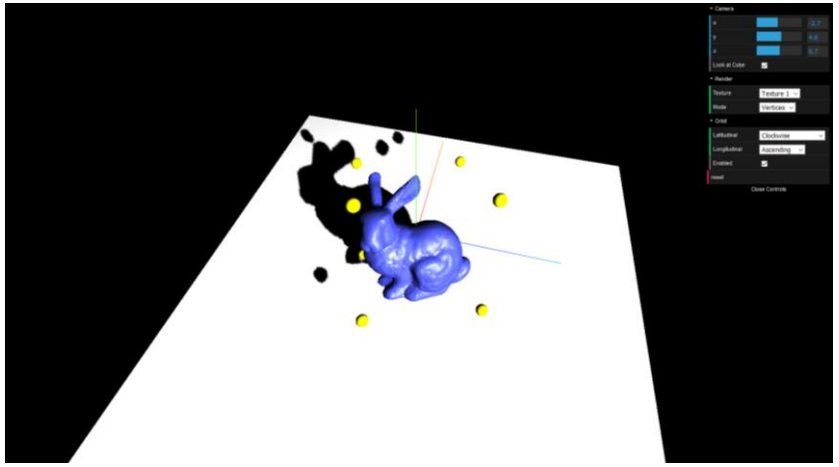

Fig. 8. Stanford Bunny with cube vertex render with camera in orbit mode.

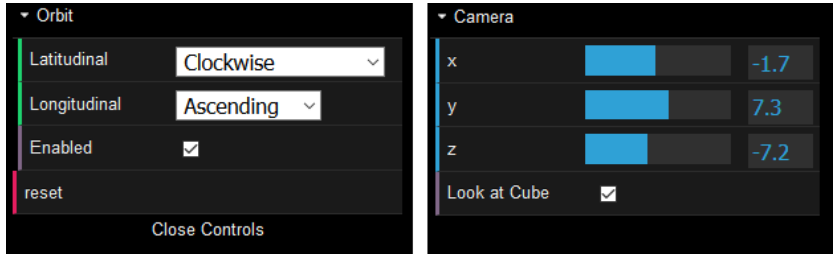

Fig. 9. A closer look at some of the control options.

III. Conclusion

Having completed the assignment, there are still many areas for improvement. The most important of these is code structure, *Three.js* and the other function libraries used require a lot of variables and parameter setting to be user properly; this structure is very different from programming languages I am familiar with. This has resulted in the solution being rather convoluted in places as functions and objects are defined, called and modified all over the place.

Another limitation is that *Three.js* is open source, this means that it is still being updated (even a new version was released during this project). This can cause software to become outdated and broken if it is not properly maintained as functions are dropped from the library. As well as this, some functions that are supposed to be stable may not work properly and it is just unfortunate if you happen to be the one to discover this.

Regarding the project, shaders would have been the next thing on the list to implement as they are highly versatile for creating custom textures. Aside from this, another interesting library is *cannon.js* which allows for the creation of objects which abide by certain physics laws with the *Three* renderer.

Going forward I would come back to use *Three.js* again as it is a simple, yet highly powerful graphical tool which this project has merely scratched the surface of.

References

[1] En.wikipedia.org. (2018). *Viewing frustum*. [online] Available at: https://en.wikipedia.org/wiki/Viewing_frustum [Accessed 09 Dec. 2018].

[2] MDN Web Docs. (2018). *window.requestAnimationFrame()*. [online] Available at: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame [Accessed 10 Dec. 2018].

[3] Tutorial.math.lamar.edu. (2018). *Calculus III - Spherical Coordinates*. http://tutorial.math.lamar.edu/Classes/CalcIII [Accessed 10 Dec. 2018].

[4] Threejs.org. (2018). three.js/documentation. [online] Available at: https://threejs.org/docs/#api/en/lights/PointLight [Accessed 12 Dec. 2018].