Introduction

This report documents the implementation of requirements of the WebGL Coursework and additional implementations. Code editing has been done on Visual Studio Code and a local webserver was used to load images for textures and meshes. To preface, some of the requirements within the coursework do crossover (example: requirement 1 and 4) thereby some additional code may be shown.

Requirement 1- Draw a simple cube.

Firstly, a *mesh* is used to draw a cube. A *mesh* contains the vertices, edges and faces that defines the shape of a polyhedral 3-D object. The mesh constructor within three.js is as shows, THREE.Mesh(*geometry*, *material*). The parameter *geometry* is for an instance of Geometry or BufferGeometry which contains the attributes positions, colors, vertex, faces, etc. The *material* parameter is for materials that describe the appearance of objects. Using the inbuilt function THREE.BoxGeometry(*width*,*height*,*depth*), I was able to create a cube geometry by inputting the width, height and depth. The material I used was given by the inbuilt function THREE.MeshPhongMaterial. This material was specifically used for my cube so that it is able to simulate shiny surfaces with specular highlights. MeshPhongMaterial will be discussed in more detail later. Given the material and geometry I was able make a mesh. Once the mesh is made it is added to the scene and the cube is drawn and is centered at origin (BoxGeometry() places the cube at origin on creation).

```
const boxGeometry = new THREE.BoxGeometry(1,1,1);
const faceMaterial = new THREE.MeshPhongMaterial( { color: '#8AC'} );
cubeFaces = new THREE.Mesh(boxGeometry, faceMaterial);
cubeFaces.castShadow = true;
cubeFaces.receiveShadow =false;
mainCube=cubeFaces;
boundingBoxCube= new THREE.Box3().setFromObject(mainCube);
cubeSize=boundingBoxCube.getSize();
scene.add( mainCube );
```

Figure 1: Code used to draw cube.

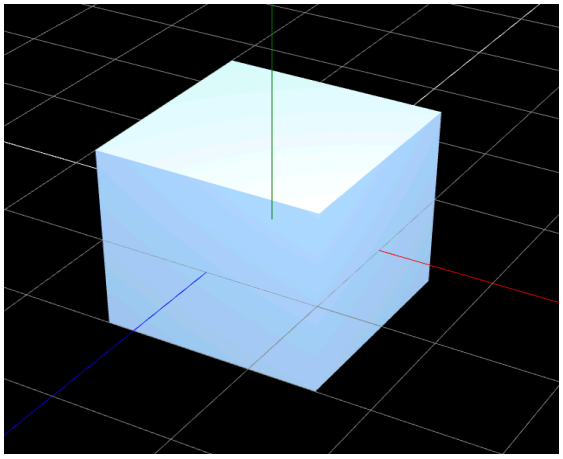

Figure 2: Cube centered at (0,0,0), faces orthogonal to, x-, y-, z- axes and with opposite corner points (–1, –1, –1) and (1, 1, 1).

Another approach found was by creating a Geometry through THREE.Geometry() where I would have to manually input my vertices position and faces. However, I went with BoxGeometry as it simpler and sufficient to satisfy the requirement.

Requirement 2- Draw coordinate system axes.

The way in which the axes are made is by drawing three lines each with different colors. The function THREE.Line(*geometry*,*material*) can be used to draw lines. A BufferGeometry is made by using the function setFromPoints which sets the attribute for the BufferGeometry from an array of points. Within three,js lines are made up of consecutive pair of vectors,hence two points are pushed into an array. One vector will always be (0,0,0) while the other vector depends on which axis will be drawn, for example the x-axis geometry may contain the vector (15,0,0). Using these two vectors the BufferGeometry can be produced. The *material* will be produced from the function THREE.LineMaterialBasicMaterial.

THREE.LineMaterialBasicMaterial allows colors of the line to be specified through the use of hex-color codes. After gaining the BufferGeometry and line material I was able to produce a line for each axis with different colors (x-axis: red, z-axis: blue, y-axis: green). Once the lines have been made, I add each line to the scene.

```
//blue line
const blueMat = new THREE.LineBasicMaterial( { color: 0x0000ff } );
const points = [];
points.push( new THREE.Vector3( 0, 0, 15 ) );
points.push( new THREE.Vector3( 0, 0, 0) );
const geometryLine = new THREE.BufferGeometry().setFromPoints( points );
const lineZ = new THREE.Line( geometryLine, blueMat );
//red line
const redMat = new THREE.LineBasicMaterial( { color: 0xff0000 } );
const points2 = [];
points2.push( new THREE.Vector3( 0, 0, 0 ) );
points2.push( new THREE.Vector3( 15, 0, 0 ) );
const geometryLine2 = new THREE.BufferGeometry().setFromPoints( points2 );
const lineY = new THREE.Line( geometryLine2 , redMat );
//green line
const points3 = [];
points3.push( new THREE.Vector3( 0, 15, 0 ) );
points3.push( new THREE.Vector3( 0, 0, 0 ) );
const geometryLine3 = new THREE.BufferGeometry().setFromPoints( points3 );
const lineX = new THREE.Line( geometryLine3 , greenMat );
scene.add( lineZ );
scene.add( lineX );
scene.add( lineY );
```

Figure 3: Code used to draw lines.

Requirement 3- Rotate the cube.

Within three.js rotations are represented through Euler Angles and Quaternions. Initially, my implementation of rotations of the objects were done by incrementing the object.rotation x, ,y, and z values as this was a simpler implementation. However, after learning more about Gimbal Lock, I realized it was not a good solution to the requirement. Gimbal lock is the loss of one degree of freedom in a three-dimensional space, it occurs when the axes of two of the three *gimbals* are within a parallel configuration. This parallel configuration can occur when using Euler Angles during rotations of the objects. Hence, I decided on the use of quaternions for the rotation of my objects. Without

getting into too much detail quaternions are “any of a set of numbers that comprise a four-dimensional vector space with a basis consisting of the real number 1 and three imaginary units i, j, k, that follow special rules of multiplication” (Merriam-Webster, n.d). Quaternions is used in computer graphics to rotate objects in three dimensions. The use of quaternions avoids gimble lock and also working with them is easier than with matrices. Through the use of multiplication of quaternions, I was able to rotate the objects. How it would work is each object would have a quaternion which would be the current orientation of it, and I would have a “rotation” quaternion. Multiplication of these two quaternions, would 1) perform the first rotation of the object quaternion on the object, 2) proceed to rotate the object again based on the “rotation” quaternion. These “rotation” quaternions are produced using the setFromAxisAngle(*axis*, *angle*) function. What this function does is that it sets a quaternion based on the *axis*, which is a vector that is normalized, and the rotation *angle* in radian. This allows me to determine how “fast” my rotation of the object will be, and which axis will my object rotate on. How rotations are displayed in the scene is through the animate function. Within the animate function there is a loop that causes the renderer to draw the scene every time it is refreshed. Placing the quaternion multiplication within the animate function will cause the rotation to continue through the loop.

```
const speed = 0.0025;
var rotateX = new THREE.Quaternion().setFromAxisAngle(new THREE.Vector3(1,0,0),speed*5);
var rotateY = new THREE.Quaternion().setFromAxisAngle(new THREE.Vector3(0,1,0),speed*5);
var rotateZ = new THREE.Quaternion().setFromAxisAngle(new THREE.Vector3(0,0,1),speed*5);
```

Figure 4: ”rotation” quaternions.

```
if(rotateCubeX==true){
const cubeQuat = mainCube.quaternion;
cubeQuat.multiplyQuaternions(rotateX,cubeQuat);
}
if(rotateCubeY==true){
const cubeQuat = mainCube.quaternion;
cubeQuat.multiplyQuaternions(rotateY,cubeQuat);
 }
if(rotateCubeZ==true){
const cubeQuat = mainCube.quaternion;
cubeQuat.multiplyQuaternions(rotateZ,cubeQuat);
```

Figure 5: Quaternion multiplication within animate function.

The ways in which I control the rotation is through the “1”, “2”, and “3” keys. How it works is that there is a global variable that acts as a Boolean. If the variable in the TRUE state, the rotation will occur. The function handleKeyDown(event) allows keyboard and mouse inputs to trigger certain events based on the specific input registered. I set keyboard inputs to start/stop rotate cube about X, Y and Z axis. This is done through keyboard inputs that change the Boolean variables. This allows the rotation about three axes at once by setting all the rotation variables to true.

```
case 49: // 1=rotate about x axis
if(rotateCubeX==false){
    rotateCubeX=true;
}else{
    rotateCubeX=false;
}
break;
case 50://2=rotate about y axis
if(rotateCubeY==false){
    rotateCubeY=true;
}else{
    rotateCubeY=false;
}
break;
case 51://3=rotate about z axis
if(rotateCubeZ==false){
    rotateCubeZ=true;
}else{
    rotateCubeZ=false;
}
break;
```

Figure 6: Shows switching of variable values.

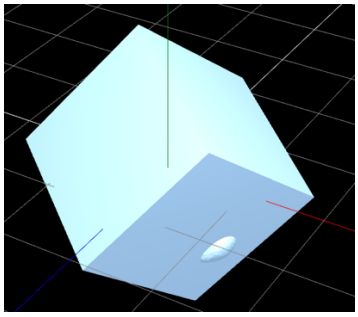

Figure 7: Cube rotation and a glimpse of the bunny.

Requirement 4-Different render modes.
The implementation of this requirement was done through the removing/adding of objects while saving the rotation of the cube when changing render modes. The way in which I implemented all three rendering modes is similar. Firstly, within the init function I created the different rendering modes (vertex, face, edges) meshes. Secondly, with keyboard inputs, I would remove the current cube on the scene and add the new specified cube on to the scene whilst maintaining orientation as the rotation is saved.

Edge render mode
To create a Cube that display the wireframe I decided to use the THREE.LineSegments(*geometry, material*) constructor. The *geometry* parameter is for a pair or multiple pairs of vertices and each of those pairs represent a line segment. The *material* parameter is for the material of the line. Within three.js there is THREE.WireframeGeometry which is used to view a geometry of an object as a wireframe. To create a geometry for our wireframe cube we must first create a BoxGeometry. Then, we make a WireframeGeometry based on the BoxGeometry through the constructor. The *material* used is the same in Requirement 2 as they are both LineBasicMaterials. With the WireframeGeometry and the line material, using the function LineSegments I was able to create a wireframe cube object and store it in a variable.

```
var wireframeGeo= new THREE.WireframeGeometry ( boxGeometry );
var greenMat = new THREE.LineBasicMaterial( { color:0x008000} );
wireFrameCube= new THREE.LineSegments( wireframeGeo,greenMat );
wireFrameCube.material.depthTest = false;
wireFrameCube.material.opacity = 10;
wireFrameCube.material.transparent = false;
wireFrameCube.castShadow = true;
wireFrameCube.receiveShadow =false;
```

Figure 8: How the wireframe cube is made.

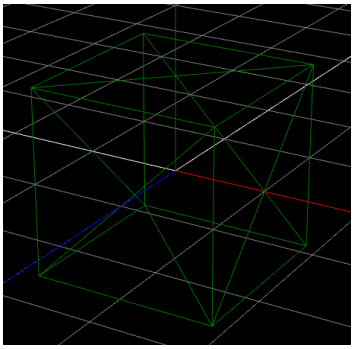

Figure 9: Wireframe cube.

Face render mode

The same cube produced as requirement one is the face cube. The *geometry* of the cube is a BoxGeometry. To simulate lighting and shading I decided to use the MeshPhongMaterial for shading and reflective attributes. The reflectance of lighting for a MeshPhongMaterial is calculated through the Blinn-Phong model. The Blinn-Phong reflectance model is also known as the modified Phong reflection model. Before talking about the Blinn-Phong reflectance, I will go through the Phong reflection model first. The Phong reflection model is based off three components: ambient, diffuse and specular.

By adding the three components the Phong lighting model is given by the equation.

$$\mathrm{P} = k_d i_d \cos\theta + k_s i_s cos^{\alpha}\varphi + k_a i_a$$

However, for multiple light source the equation is slightly altered.

$$\mathrm{P} = k_d \sum_i i_{d,i} \cos\theta_l + k_s \sum_i i_{s,i} cos^{\alpha}\varphi_l + k_a i_a$$

i – light source index
$i_{d,i}$ – light source $i's$ intensity for diffuse
$i_{s,i}$ – light source $i's$ intensity for specular
θ_l/φ_l – angles for light source i

For computation in a computer, it is better to use vector operations instead of cosine operations as they are much more efficient. Hence the following equation is formed.

$$\mathrm{P} = k_d \sum_i i_{d,i}(L_i \cdot N) + k_s \sum_i i_{s,i}(R_i \cdot V)^{\alpha} + k_a i_a$$

Where k_d, k_a, k_s corresponds to the diffuse, ambient and specular reflectivity of the object material.
$i_{d,i}$, $i_{s,i}$, i_a corresponds to the diffuse, reflectivity and ambient of a light source i.
L is the unit vector pointing to the light source.
N is the surface normal.
R_i is the reflection direction of the light.
V is the viewing direction.
a is the shininess of the object's material.

However, the Blinn-Phong reflectance model differs in a way such that, instead of calculating the dot product between R_i and V continuously it calculates the halfway vector H between the viewer V and the Light Source L.

$$H = \frac{L + V}{\|L + V\|}$$

Thus, the dot product $(R_i \cdot V)$ is replaced with $(N \cdot H)$ within the equation.

Together with Phong shading and Blinn-Phong reflectance, objects that have the MeshPhongMaterial and have lights on them would display properties shown in Figure 2, 7 and 10.

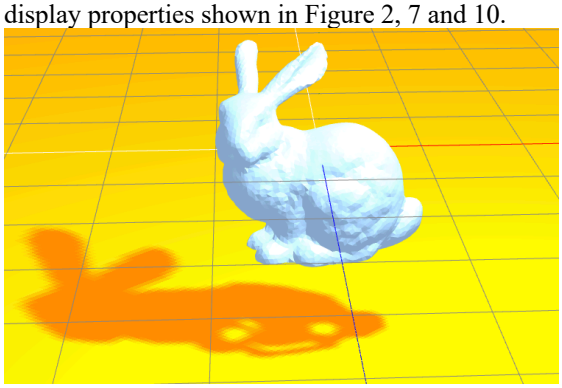

Figure 10: Example of MeshPhongMaterial on a bunny.

Vertices render mode

The way in which this implementation was approached is through using the three.js class THREE.Points. The constructor Points(*geometry*,*material)* works similarly to any other constructor. Using the BoxGeometry and PointsMaterial I was able to construct points placed at the 8 vertices postitions.

```
let pointsMaterial = new THREE.PointsMaterial( { size: 0.06, color: 0xff0000 } );
cubeVertex = new THREE.Points(boxGeometry,pointsMaterial);
```

The other approach I initially thought of was iterating through the vertices within the geometry of the box and adding a sphere centered at each vertex. However, I later discovered it was not a suitable implementation when an object has a large number of vertices. Which caused me to pivot towards using Points.

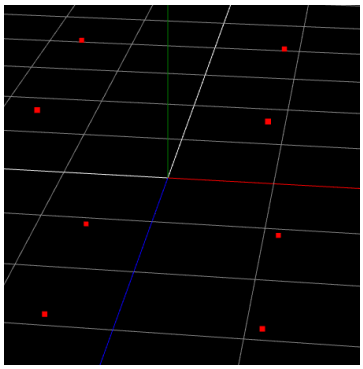

Figure 11: Vertex render mode of cube.

Rotation while switching render modes.
While the cube is rotating and the render modes are changed, the rotation of the main cube is saved, the main cube is removed from the scene, a new cube will be added into the scene with the initial orientation set using the saved rotations. This will make the rotation seem fluid when switching between render modes.

Requirement 5- Translate the camera.
Within three.js the camera is an object. Functions translateX, translateY and translateZ; translates an object along the x, y and z axis in the object space. This allows us to move the camera along it's x, y and z axis.

```
case 37://left key
    saveX=camera.position.x;
    saveY=camera.position.y;
    saveZ=camera.position.z;
    camera.translateX(-0.5);
    diffX=camera.position.x-saveX;
    diffY=camera.position.y-saveY;
    diffZ=camera.position.z-saveZ;
    lookAtCoor.x=lookAtCoor.x+diffX;
    lookAtCoor.y=lookAtCoor.y+diffY;
    lookAtCoor.z=lookAtCoor.z+diffZ;
    break;
```

Figure 12: Translating the camera along its x-axis and tracking of look at coordinates.

Mathematically, as the position vector contains the object's local position, addition/subtraction of the position vector causes it to move along its left, right, up, down back and forward vectors; this is what translateX, translateY and translateZ does. When the camera is moved the look at coordinate is also moved. The camera's position is measured before and after the moving and the difference is used to change the look at coordinate.

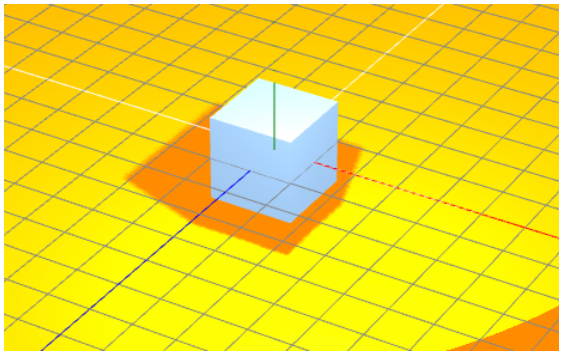

Figure 13: The camera is moved back and upwards, the cube gets smaller and see more of the grid.

Requirement 6- Orbit the camera.
In order to replicate the orbit controls found in three.js's OrbitControl I decided to use spherical coordinate system. Within the spherical coordinate system, a position of a point is represented by the azimuthal angle, a polar angle and the radial distance of that point from a fixed origin.

the radial distance ρ

the azimuthal angle θ

the polar angle ϕ

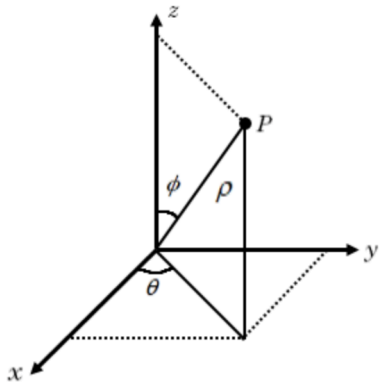

Figure 14: Spherical Coordinate System representation. (Mathworks,n.d).

The ability to orbit horizontally can be seen by converting the cartesian coordinates given in three.js into spherical coordinates, changing the azimuthal angle and converting back to cartesian coordinates. Similarly, for vertical orbit the cartesian coordinates are converted into spherical coordinates, change the polar angle and convert it back to cartesian coordinates. Once you have done the conversions the camera position is then set to the new cartesian coordinates.

However, notice that within the Fig. 13 the z, y and x- axis are at different positions than in three.js, hence the following equations would need to be slight altered as well.

Converting from cartesian to spherical coordinates.
To get a point in the spherical coordinates we need a radial distance. This radial distance is actually saved when the camera is first added to the scene. The distance between the look at point and the camera will be the same throughout the scene hence there is no need to calculate it. Next, we require the polar angle, to obtain the polar angle we may use the equation,

$$polar\ angle = \arccos\left(\frac{y}{radial\ distance}\right)$$

and for the azimuthal angle

$$azimuthal\ angle = \arctan\left(\frac{x}{z}\right).$$

There are exceptions when the cartesian coordinates of the points are negative that must be considered. If the y is negative the azimuthal angle would be:

$$polar\ angle = \pi - \arccos\left(\frac{|y|}{radial\ distance}\right)$$

If the x coordinate is negative the equation would be:

$$azimuthal\ angle = -1 * \arctan\left(\frac{|x|}{z}\right)$$

If the z coordinate is negative the equation would be:

$$azimuthal\ angle = \pi - \arctan\left(\frac{x}{|z|}\right)$$

If both x and z is negative the azimuthal angle would be:

$$azimuthal\ angle = \arctan\left(\frac{|x|}{|z|}\right) - \pi$$

Since the spherical coordinate is based on the origin (0,0,0) it needs to alter it so that the new "origin" would be the look at point. How this is done is by deducting the position of the camera with the look at point.

```
var x=camera.position.x-lookAtCoor.x;
var y=camera.position.y-lookAtCoor.y;
var z=camera.position.z-lookAtCoor.z;
```

Figure 15: Changing the coordinates to fit the new origin.

"Moving" the camera
Once I've obtained the angles required, depending on horizontal or vertical orbit, I would change either angle by addition or subtraction with a radian angle which I set as variable t and set it as 0.1.

Converting from spherical to cartesian coordinates.
Once subtracted/added I would then convert the spherical coordinates back to cartesian coordinates through the equations:

$$z = radial\ distance * \sin(polar) * \cos(azimuthal)$$
$$x = radial\ distance * \cos(polar)$$
$$y = radial\ distance * \sin(polar) * \sin(azimuthal)$$

Then we would need the add back the look at point coordinates that we removed earlier to x, y, z.

$$z = z + lookAtCoor.z$$
$$x = x + lookAtCoor.x$$
$$y = y + lookAtCoor.y$$

Setting camera look at after rotation
Finally, we set the new position to our camera. The look at point of the camera must be set again so that during orbit it will focus on the look at point. This is done through camera.lookAt(x,y,z) function.

```
case 77: //m key-orbit camera to the right
//find cartesian coordinates
    var x=camera.position.x-lookAtCoor.x;
    var y=camera.position.y-lookAtCoor.y;
    var z=camera.position.z-lookAtCoor.z;
    //convert cartesian coordinates->spherical coordinates
    if(y<0){
        angle= Math.PI-Math.acos(Math.abs(y)/dist);
    }else{
        angle = Math.acos(y/dist);
    }
    if(z<0 && x<0){
        gamma=Math.atan(Math.abs(x)/(Math.abs(z)))-Math.PI;
    }else if(z<0){
        gamma = Math.PI - Math.atan(x/(Math.abs(z)));
    }else if(x<0){
        gamma = -1* Math.atan(Math.abs(x)/z);
    }else{
        gamma= Math.atan(x/z);
    }
    //change angle for orbit of camera
    gamma+=t;
    //convert spherical->cartesian coordinates
    camera.position.z=dist*Math.sin(angle)*Math.cos(gamma)+lookAtCoor.z;
    camera.position.y=dist*Math.cos(angle)+lookAtCoor.y;
    camera.position.x=dist*Math.sin(angle)*Math.sin(gamma)+lookAtCoor.x;
    camera.lookAt(lookAtCoor);
    break
```

Figure 16: Code snippet of horizontal rotation to the right. (leftwards orbit is similar)

```
case 75: //k key-orbit camera vertically downwards
//find cartesian coordinates
    var x=camera.position.x-lookAtCoor.x;
    var y=camera.position.y-lookAtCoor.y;
    var z=camera.position.z-lookAtCoor.z;
    //convert cartesian coordinates->spherical coordinates
    if(y<0){
        angle= Math.PI-Math.acos(Math.abs(y)/dist);
    }else{
        angle = Math.acos(y/dist);
    }
    if(z<0 && x<0){
        gamma=Math.atan(Math.abs(x)/(Math.abs(z)))-Math.PI;
    }else if(z<0){
        gamma = Math.PI - Math.atan(x/(Math.abs(z)));
    }else if(x<0){
        gamma = -1* Math.atan(Math.abs(x)/z);
    }else{
        gamma= Math.atan(x/z);
    }
    //change angle for orbit of camera
    angle+=t;
    //convert spherical->cartesian coordinates
    camera.position.z=dist*Math.sin(angle)*Math.cos(gamma)+lookAtCoor.z;
    camera.position.y=dist*Math.cos(angle)+lookAtCoor.y;
    camera.position.x=dist*Math.sin(angle)*Math.sin(gamma)+lookAtCoor.x;
    camera.lookAt(lookAtCoor);
    break;
```

Figure 17: Code snipper of vertical rotation downwards(upwards orbit is similar)

Orbiting Horizontally worked well, however there was in convenience when orbiting vertically. For example, when the camera is orbiting upwards

vertically, as it crosses the *y* axis you would have to switch keys from moving upwards to downwards to be able to do a proper vertical rotation around the look at point. While testing the orbits there was no sign of gimbal lock. How I tested it was by orbiting the camera vertically until I face the top of the cube and orbit horizontally left and right.

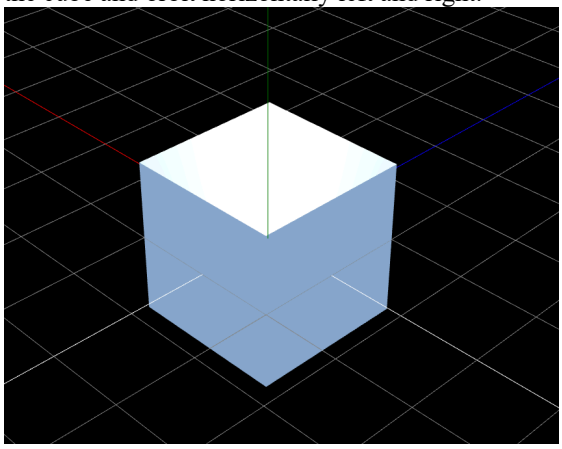

Figure 18: shows the camera behind the cube by horizontal orbit with look at point (0,0,0).

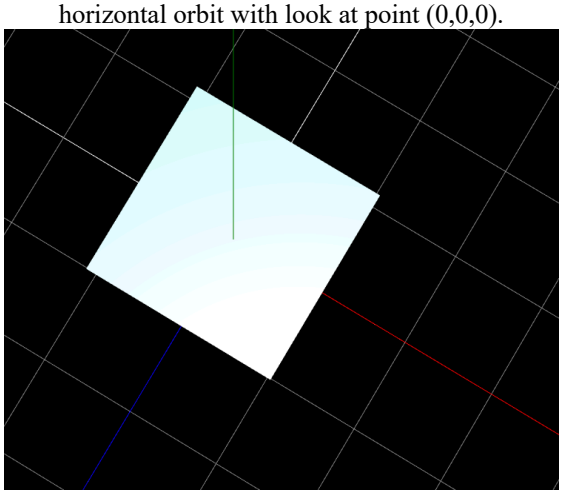

Figure 19: shows the camera above the cube by vertical orbit with look at point (0,0,0).

Requirement 7: Texture mapping.

To wrap textures on to the different sides of the cube I first had to load the textures. I decided to use the loader THREE.TextureLoader. This allowed me to load the pictures that would later be applied to my cube. How it works is that the pictures will be loaded into a material array and be set as a material. I decided to make the textures a PhongMaterial. A mesh will then be created with BoxGeometry and the material array.

```
var loader = new THREE.TextureLoader();
var materialArray = [
    new THREE.MeshPhongMaterial( { map: loader.load("http://localhost:8000/Downloads/CM20219/
    new THREE.MeshPhongMaterial( { map: loader.load("http://localhost:8000/Downloads/CM20219/
    new THREE.MeshPhongMaterial( { map: loader.load("http://localhost:8000/Downloads/CM20219/
    new THREE.MeshPhongMaterial( { map: loader.load("http://localhost:8000/Downloads/CM20219/
    new THREE.MeshPhongMaterial( { map: loader.load("http://localhost:8000/Downloads/CM20219/
    new THREE.MeshPhongMaterial( { map: loader.load("http://localhost:8000/Downloads/CM20219/
];
texturedCube = new THREE.Mesh( boxGeometry, materialArray );
texturedCube.castShadow = true;
texturedCube.receiveShadow =false;
```

Figure 20: Code to load textures and create textured cube.

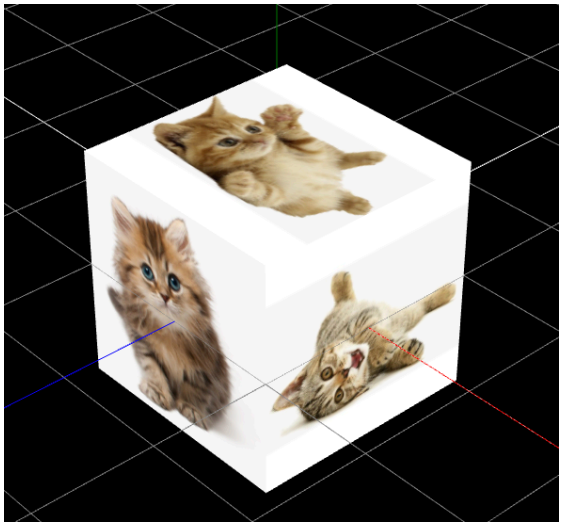

Figure 21: Textured cube three faces.

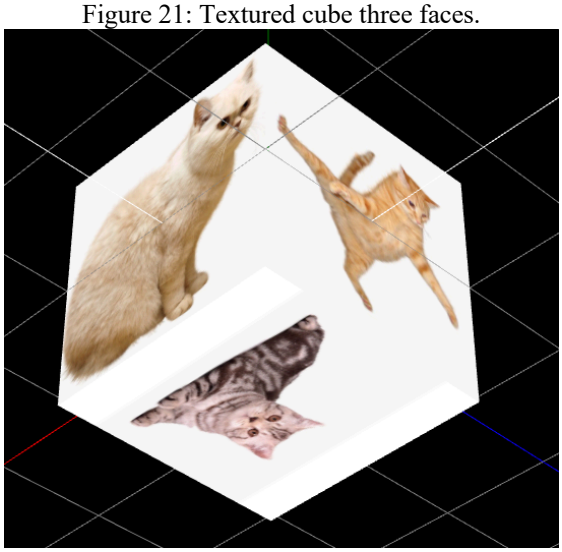

Figure 22: Textured cube other three faces.

The textures loaded required a size of image of power of two. So, to prevent skew I altered the images to make the sizes a power of two.

Requirement 8- Loading a mesh object.

To load the bunny, I used the loader OBJLoader, this allows me to load an .obj file. An .obj file consist of a geometry of a given object. The attributes it may contain are position of each vertex, vertex normals, the faces which consist of list of vertices, texture vertices, etc. Within the 'bunny-5000.obj file' it contains the 5000 vertices and 9906 faces. The OBJLoader will then go through the file and create the bunny using the given vertices and faces. With just adding the object to the scene the bunny will be white as there isn't any material on it. How I approached this requirement is by using the geometry from the object and making a new mesh with the material I

require. This is similar to how I approached requirement 4.

Scaling the bunny
To fit the bunny inside the cube, I first had to figure out the scale factor between the cube and the bunny. How I did this was through making a bounding box from the bunny and the cube. Using the getSize function I was able to obtain a size vector for both bounding boxes. To determine the ratio of the vectors I divided the size vector of both bounding boxes. From the new ratioed vector I can obtain the scale factor and scale the bunny down using the scale.set(x,y,z) function.

Setting the bunny position.
Using the bounding boxes, I was able to determine the center of the now scaled bunny. Once I found the center of the scaled bunny, I just found the distance between the center of the cube (0,0,0) and the center of the bunny. The difference is then placed into the position.set(x,y,z) function for the bunny and the model is moved.

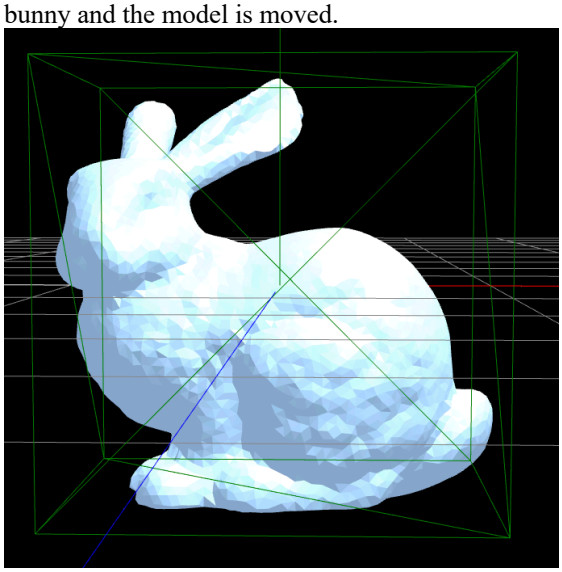

Figure 23: The bunny fits perfectly into the cube.

Requirement 9-Rotate the mesh and render different modes.
The implementation to this requirement is very similar to the implementation of requirement 3 and 4. I first loaded the bunny and made multiple meshes each for the different rendering modes and stored them in a global variable. Then with key presses I was able to switch between render modes while keeping rotation continuous.

Rotation of bunny
Rotation for the bunny is done the same way as the cube, through the use of quaternions.

Faces mode of bunny
The bunny geometry is used as the mesh's geometry again. The material used was the same to the cube face render mode. This made the bunny have reflections and shaders as it was a Phong Material. The bunny can be seen at figure 23.

Vertex mode of bunny
The vertex mode of the bunny was created by using THREE.Points again. By using the bunny geometry and the use of THREE.PointsMaterial the positions of the bunny's vertices will be marked with red color points. The initial implementation I had by placing sphere objects at the bunny's vertices made the scene drop FPS hence the reason why I switched approaches. Also, it was a much easier implementation using THREE.Points.

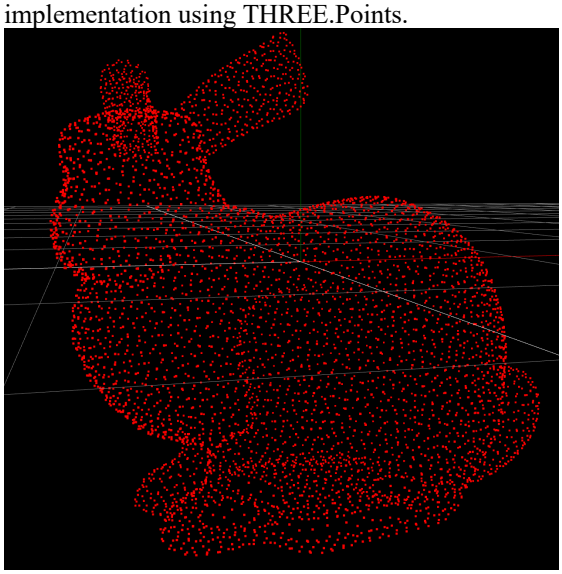

Figure 24: Bunny vertices render mode

Wireframe mode of bunny
The wireframe mode of bunny was done by making a WireframeGeometry out of the bunny geometry. WireframeGeometry does accept Buffered Geometries so the bunny's geometry was accepted.

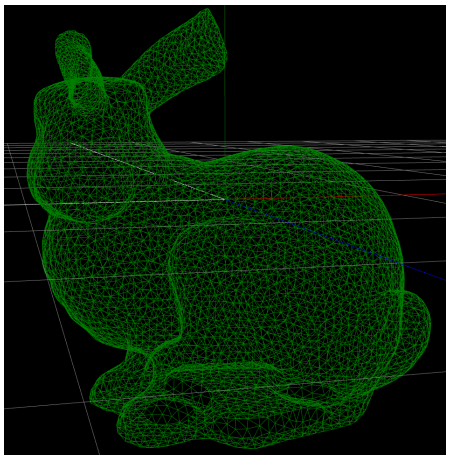

Figure 25: Bunny wireframe render mode

```
objloader.load(
    //remember to change the path to these objects as it varries from pc to pc
        'http://localhost:8000/Downloads/CM20219/bunny-5000.obj',
        //Load wireframe render bunny
        function ( object ) {
            object.traverse( function ( child ) {
            if (child.isMesh) {
            //to find scale factor of cube and bunny
            boundingBoxBunny = new THREE.Box3().setFromObject(child);
            bunnySize=boundingBoxBunny.getSize();
            var scaleFactorX = cubeSize.x / bunnySize.x;
            var scaleFactorY = cubeSize.y / bunnySize.y;
            var scaleFactorZ = cubeSize.z / bunnySize.z;
            const wireframeGeometry = new THREE.WireframeGeometry(child.geometry);
            const wireframeMaterial = new THREE.LineBasicMaterial({color: 0x008000});
            const wireframe = new THREE.LineSegments(wireframeGeometry, wireframeMaterial);
            bunnyWireframe=wireframe;
            bunnyWireframe.scale.set(scaleFactorX,scaleFactorY,scaleFactorZ);
            var scaledBoundingBoxBunny = new THREE.Box3().setFromObject(bunnyWireframe);
            var center = scaledBoundingBoxBunny.getCenter();
            //cube center is 0,0,0
            //find difference in center
            //set position of bunny based on distance
            var posiFactorX = 0-center.x;
            var posiFactorY = 0-center.y;
            var posiFactorZ = 0-center.z;
            bunnyWireframe.position.set(posiFactorX,posiFactorY,posiFactorZ);
            //Load vertex render bunny
            const material = new THREE.PointsMaterial( { size: 0.01, color: 0xff0000 } );
             var points2 = new THREE.Points(child.geometry,material);
             points2.scale.set(scaleFactorX,scaleFactorX,scaleFactorX);
             points2.position.set(posiFactorX,posiFactorY,posiFactorZ);
             bunnyVertex=points2;
            //Load faces render bunny
            bunnyFaces= new THREE.Mesh(child.geometry,faceMaterial);
            bunnyFaces.scale.set(scaleFactorX,scaleFactorX,scaleFactorX);
            bunnyFaces.position.set(posiFactorX,posiFactorY,posiFactorZ);
            bunny=bunnyFaces;
            bunny.castShadow = true;
            bunny.receiveShadow =false;
            scene.add(bunny);
            }
            } );
        });
```

Figure 26: code snippet of producing different bunny render modes

Requirement 10- Be creative.

Playing with lighting

Within this requirement I firstly tried playing with the different kinds of lighting to make my objects look good when I took the screenshots for report. Through the different lighting I preferred the spotlight lighting or the point light. I stuck with spotlight as it looked the best with the orange plane.

Loading 3-D models

The orange plane I placed below the objects were to test out how shadows interact with different objects, motion and also lighting. I tried adding some GLTF files, but I could not seem to get the GLTFloader to work due to different three.js versions. Since I could not get the GLTF to load I decided to test out more .obj files. So far, I've only loaded two more objects downloaded from a website, a wolf and a cat (snippysnappets,2017), for some reason I couldn't make the other objects produce shadows.

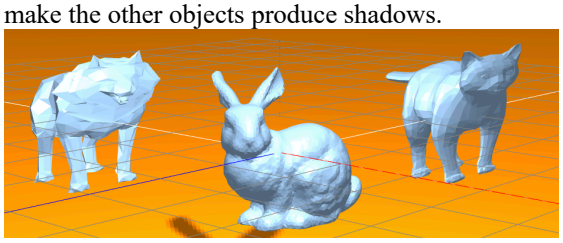

Figure 27: A bunny, wolf and cat.

Animating bunny hops

Instead of my bunny just rotating, I wanted to see it hop. That's when I tried anime the bunny to hop from one position to another. Using trigonometry, I was able to create a nice curve hop but my bunny was not able to hop in random directions like I hoped. The bunny also does not hop from its initial position which is a bug I have yet to fix.

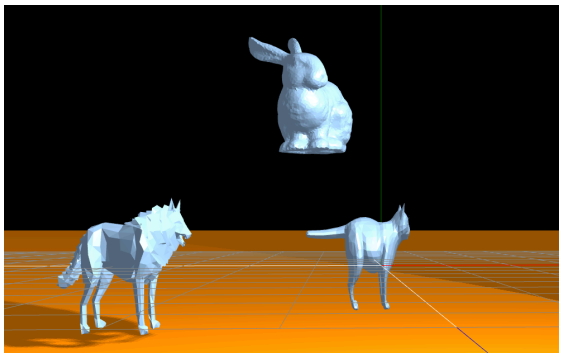

Figure 28: A bunny mid hop.

Conclusions

The overall coursework has brought forth a new way of looking at 3-D graphics and 3-D animation. If I had more time, I would definitely try to load more 3-D models and attempt to load materials so that my objects will look better. Animation was what piqued my interest during this coursework so I would also like to see more of how smooth and natural representations of motion can be produce within three.js. It was difficult to get a grasp on how good lighting is made so more time will definitely be spent there.

References

Mathworks, n.d. *Plotting in Spherical Coordinates System*. Available from: https://uk.mathworks.com/help/symbolic/plotting-in-spherical-coordinate-system.html [Accessed 16 December 2020].

Merriam-Webster, n.d. *Quaternion|Definition of Quaternion by Merriam-Webster* [Online] Merriam-Webster. Available from: https://www.merriam-webster.com/dictionary/quaternion [Accessed 12 December 2020].

snippysnappets, 2017. *Low Poly Wolf 3D Model*[Online]. Available from: https://free3d.com/3d-model/low-poly-wolf-4601.html [Accessed 16 December 2020].

snippysnappets, 2017. *Low Poly Cat 3D Model* [Online]. Available from: https://free3d.com/3d-model/low-poly-cat-46138.html [Accessed 16 December 2020].