CM20219 COURSEWORK PART 2: REPORT

As part of this coursework I got to explore how WebGL works and apply it using the three.js Application Programming Interface. This document summarizes what I have done, and offers some explanations and discussions surrounding the work. I have referred to the three.js documentation extensively during the development to get a better understanding of the library and what it offers [1].

Requirement 1 – Draw a simple cube

The first requirement was simple to implement. Only a few lines of code created a cube that fits the requirements of the marking scheme.

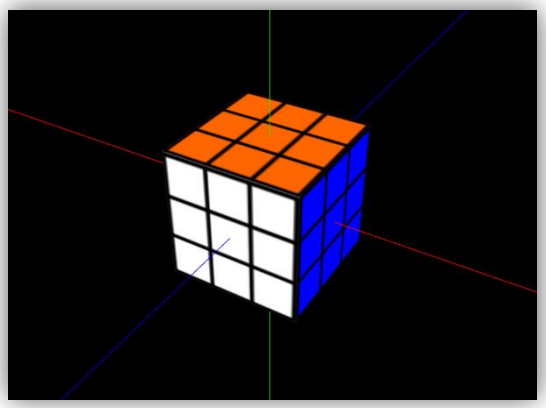

Figure 1: Screenshot of the cube is as it's initially loaded in.

```
var geometry = new THREE.BoxGeometry(2, 2, 2);

var loader = new THREE.TextureLoader();

var texture = loader.load('cube.jpg');
textureMaterial = new THREE.MeshPhongMaterial({
      map: texture
});


cube = new THREE.Mesh(geometry,
textureMaterial);
scene.add(cube);
```

The `geometry` variable uses a three.js function to create a simple cube geometry of size 2. The `texture` variable contains the texture that is applied to the cube. How this works is covered in the section of the report that discusses requirement 7. The texture is then applied to a MeshPhongMaterial which is a material that calculates the Phong shading for itself when lit up by light (however not an `AmbientLight`, as it does "not have a direction" [7]). The three.js mesh class combines the geometry and textureMaterial to create an actual object, which can be seen in Figure 1. The created object is by default positioned such that its centre is at the origin.

Given more time I would certainly look more into different types of geometries and how that helps represent different objects

Requirement 2 – Draw coordinate system axes

Drawing the coordinate axes was just as straightforward as drawing the cube, although it required some more code.

```
var xLineMaterial = new THREE.LineBasicMaterial({color: 0xff0000});

var xGeometry = new THREE.Geometry();
xGeometry.vertices.push(new THREE.Vector3(-10, 0, 0));
xGeometry.vertices.push(new THREE.Vector3(10, 0, 0));

var xLine = new THREE.Line(xGeometry, xLineMaterial);
```

NB: the code snippet only shows implementation for the x axis, the other axes were implemented in the same way, but the code was omitted from this document.

The principles behind creating a line are the same as for a cube. First a material is created, this time is a LineBasicMaterial which is defined as a material for "drawing wireframe-style geometries" by the documentation [2]. In the constructor, that material is given a colour represented by a hexadecimal value. Then a geometry is created, and two vertices are added to it such that a line is created between them. The line cannot be infinite therefore I have chosen to make it have length 20.

Finally, a line object is created from the geometry and material. The documentation states that the line is rendered using the WebGL function `gl.LINE_STRIP` [3]. The created lines can be seen in Figure 1.
To further develop this implementation, I would like to try and see if there is a way to create an infinite line. Dynamically rendering a line of appropriate length such that it fills the whole field of view of the camera could be a solution.

Requirement 3 – Rotate the cube

To rotate the cube (and the bunny object, see section 9 for more information about that) I have used the property of `Object3D`, `rotation`. This property is the "object's local rotation" in radians [4]. Modifying the rotation vector makes the object rotate in its own axis, not in the world axis. The rotation is toggled by a Boolean value that is toggled when pressing certain buttons: "x" for a rotation in the x axis, "y" for y axis, and "z" for z axis. If the Boolean is true, then the snippet of code above runs. Also, if the bunny is present (`bunnyMode` is true) then the bunny rotates alongside the cube.

```
if(rotateX){
        cube.rotation.x += rotation;
        if(bunnyMode){
                bunny.rotation.x += rotation;
        }
}
```

The `rotation` vector is a vector of Euler angles. The three values in the vector represent the orientation of the object, and are used to rotate the object. The problem with Euler angles is that they cause Gimbal lock. Rotating in the y axis by $\frac{\pi}{2}$ causes the z and x axis rotations to add up, which is a typical problem associated with Gimbal locking. Given more time I would have liked to look more deeply into quaternions, as they do not suffer from Gimbal lock [5], and try and use them to rotate the object. It is a more difficult representation to understand but it is more powerful, and proves useful in other applications as well e.g. orbiting objects around other objects.

Given a vector of Euler angles $E = [x, y, z]^T$ converting it to a quaternion Q is completed using the following equation [6]:

$$Q = \begin{bmatrix} \cos\left(\frac{x}{2}\right) \\ 0 \\ 0 \\ \sin\left(\frac{x}{2}\right) \end{bmatrix} \begin{bmatrix} \cos\left(\frac{y}{2}\right) \\ 0 \\ \sin\left(\frac{y}{2}\right) \\ 0 \end{bmatrix} \begin{bmatrix} \cos\left(\frac{z}{2}\right) \\ \sin\left(\frac{z}{2}\right) \\ 0 \\ 0 \end{bmatrix}$$

This can then be applied to the object to rotate it correctly without Gimbal lock issues.

Requirement 4 – Different render modes

The change of cube render mode is triggered by pressing keyboard keys. "v" for vertex mode, "e" for edge mode, and "f" for face mode. Figure 1 in section 1 depicts face mode of the cube, the other two modes are shown in Figures 2 and 3.

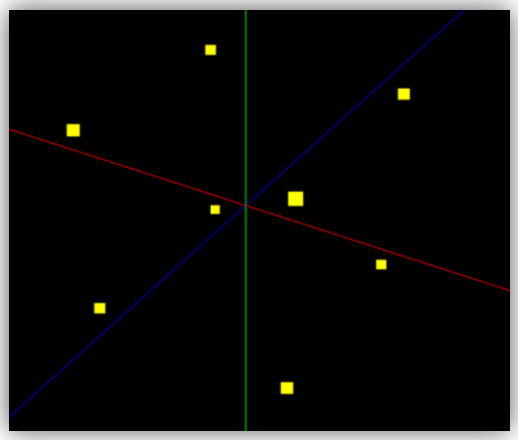

Figure 2: Vertex mode of the cube.

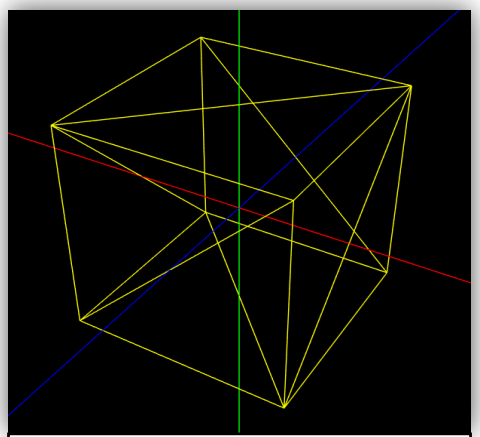

Figure 3: Edge mode of the cube.

Pressing each button does different things depending on if `bunnyMode` is true. In this requirement I will only focus on what happens when `bunnyMode` is false, the other case is discussed in section 9 of the document.

The first code snippet on the left shows the general case of what happens when a button is pressed, and also how the material of the cube is modified for the face rendering mode. In each rendering mode firstly the cube is removed from the scene. Then the geometry of the cube is copied into a variable. Then the material is modified, and a new cube created using the geometry and material.

```
scene.remove(cube);
var geometry = cube.geometry;
cube = new THREE.Mesh(geometry, textureMaterial);
scene.add(cube);
```

```
var material = new THREE.MeshBasicMaterial({
      color: 0xffff00, // yellow
      wireframe: true
});
```

```
var pointsMaterial = new THREE.PointsMaterial({
      transparent: false,
      size: 0.25,
      color: 0xffff00
});
cube = new THREE.Points(geometry, pointsMaterial);
```

For the face rendering mode, the premade `textureMaterial` is used to texture the cube. For the edge rendering mode the `wireframe` property of `Object3D` is used. Firstly a new cube is made, and in the constructor the wireframe property is set to true. This property displays all the edges of the object, that is edges of all triangles making up the object. I chose to highlight them in yellow, just to make them stand out from the axis lines. The vertex mode uses a different type of material, a `PointsMaterial`, to display the points. That material and the cube geometry are used to create a `Points` object, which is the standard way to display points in three.js. The special type of material for this type of object is required for it to work, any other material type would not work.

Requirement 5 – Translate the camera

Translating the camera is done using six keyboard keys, the four arrow keys and the numpad plus and minus buttons. Pressing one of the buttons triggers the `translateCamera` function.

```
function translateCamera(camera, changeInPosition, axis){
      var cameraPositionBefore = camera.getWorldPosition();
            switch (axis){
                  case "z":
                        camera.translateZ(changeInPosition);
                        break;
                  case "x":
                        camera.translateX(changeInPosition);
                        break;
                  case "y":
                        camera.translateY(changeInPosition);
                        break;
            }
      var cameraPositionAfter = camera.getWorldPosition();
      lookAt.add(cameraPositionAfter.sub(cameraPositionBefore));
}
```

The function takes three inputs: the camera that is meant to be translated, the amount of translation, and the axis in which to translate the camera. The function then translates the camera in the appropriate axis by the appropriate amount. The three axis translation functions used to translate the camera (`translateX`, `translateY`, and `translateZ`) are inherited from the `Object3D` class.

The `translateCamera` function also updates the global `lookAt` vector according to the change in world position of the camera. It stores the position of the camera before the translation using the `getWorldPosition` function which "returns a vector representing the position of the object in world space" [4], then it subtracts that from the position of the camera after translation. It then adds that to the `lookAt` vector, updating its position. This is done so that the camera can correctly orbit around the point. Simply translating the `lookAt` point in the same axis by the same distance would not work as the camera translates along its own coordinate system rather than the world one, whereas the `lookAt` point is a point on the world coordinate system. Any rotation of the camera would cause the translation of the point to be off.

Requirement 6 – Orbit the camera

Orbiting the camera was a slightly more involved task. It required the coordinates of the camera to be translated into spherical coordinates, the spherical coordinates changed, and then the spherical coordinates changed back into Cartesian coordinates and applied to the camera to translate it appropriately. The camera can be orbited latitudinally using the "u" and "j" keys, and longitudinally using the "h" and "k" keys.

```
function orbit(changeInTheta, changeInPhi)
{
	var cameraPosition = new THREE.Vector3();
	cameraPosition.subVectors(camera.position, lookAt);

	var radius = camera.position.distanceTo(lookAt);
	var theta = Math.acos(cameraPosition.getComponent(1)/radius);
	var phi = Math.atan(cameraPosition.getComponent(2)/cameraPosition.getComponent(0));

	phi += changeInPhi;
	if(cameraPosition.getComponent(0) < 0){
		phi += Math.PI;
	}

	theta += changeInTheta;
	if (theta > Math.PI){
		theta = Math.PI;
	} else if (theta < 0){
		theta -= changeInTheta;
	}

	var x = radius * Math.sin(theta) * Math.cos(phi) + lookAt.x;
	var z = radius * Math.sin(theta) * Math.sin(phi) + lookAt.z;
	var y = radius * Math.cos(theta) + lookAt.y;

	camera.position.set(x, y, z);
	camera.lookAt(lookAt);
}
```

The whole function above bases itself on the spherical coordinate system definition obtained from Wikipedia [8]. In that system, the position of a point within a three-dimensional space is defined by three values: the **radial distance** r from the origin, the **polar angle** θ from a fixed zenith direction, and the **azimuth angle** ϕ measured as the angle from an axis on a plane orthogonal to the zenith and passing through the origin. In Figure 4, the zenith direction is the z axis, and the azimuth angle is measured from the x axis on the x-y plane.
Some modifications had to be made to this due to how the default up direction is defined by three.js. The default up unit vector is $(0, 1, 0)$, which defines the up direction as the y axis [4]. Therefore, in the function above, and any discussion below, the z and y axes are swapped in comparison to Figure 4.

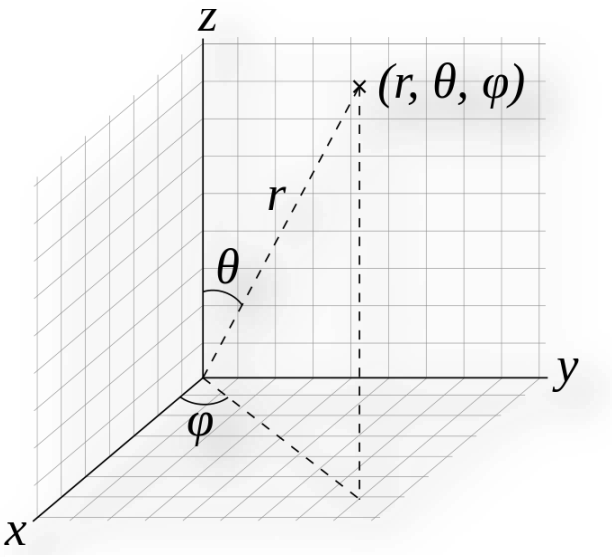

Figure 4: Diagrammatical depiction of the spherical coordinate system, obtained from [8]

Before converting the Cartesian coordinates of the camera into spherical ones the `cameraPosition` vector is calculated, which is the relative position of the camera to the `lookAt` vector. This is so that the angles are computed correctly when converting to spherical coordinates.

Then the conversion to spherical coordinates is completed using the following equations:

$$r = \sqrt{x^2 + y^2 + z^2} \qquad \theta = \arccos\left(\frac{y}{r}\right) \qquad \phi = \arctan\left(\frac{z}{x}\right)$$

where (x, y, z) are all components of the `cameraPosition` vector.

NB: in the code I use the premade function `distanceTo` to compute r however that function uses Pythagoras' Theorem to compute the distance [9] so I defined the calculation of the radius based on that.

After doing this I can freely modify the two angles θ and ϕ in order to orbit the camera in latitude and longitude respectively. However, the two angles are constrained by the range of solutions to the two trigonometric equations, the principal values of the equations. This causes θ to be constrained for values between 0 and π: $0 \leq \theta \leq \pi$. ϕ is constrained by the principal values of arctan: $-\frac{\pi}{2} \leq \phi \leq \frac{\pi}{2}$ [10].
For θ this does not cause any obvious problems until the value of the angle reaches π, then adding a value that makes the angle go beyond π seems to cause the camera position to mirror itself in the x axis. The same happens for when θ is below zero. Therefore, I have constrained θ to be within the principal values, although it cannot be equal to zero as that causes a bug that positions the camera at $(0, 1, 0)$ and causes it to be immovable. Therefore, I constrain θ to not go below zero but also not be equal to zero.
In terms of ϕ, the principal values cause the rotation to be limited to positive values of x. So for all $x < 0$ I set $\phi = \phi + \pi$.

Lastly the three computed spherical coordinates are converted back into cartesian coordinates using the following equations [8]:

$$x = r * \sin(\theta) * \cos(\phi), y = r * \cos(\theta),\ z = r * \sin(\theta) + \sin(\phi).$$

All these get the respective `lookAt` component added to them to translate the camera appropriately. The cartesian coordinates are then set as the position of the camera, and the camera is made to look at the `lookAt` point.
Given more time I would like to explore how to make this work using mouse controls rather than keyboard presses, as the mouse seems like a more intuitive input device to use when controlling an arc ball camera.

Requirement 7 – Texture mapping

I have implemented texture mapping using UV mapping. UV mapping is a process of applying a 2D texture, a texture map, onto a 3D object [11]. This is done in three steps.
First the texture is loaded into a material using a `TextureLoader`, and set as the `map` property of the material.
Then, as seen in the code snippet on the right, an array of coordinates is created that represents the four corners of one face. The coordinates are in the UV format, so they have to be between zero and one. The coordinates for the other five faces are defined in the same way.

```
var face1 = [
        new THREE.Vector2(0, .666),
        new THREE.Vector2(.5, .666),
        new THREE.Vector2(.5, 1),
        new THREE.Vector2(0, 1)
];
```

```
geometry.faceVertexUvs[0][0]=[face1[0],face1[1],face1[3]];
```

Then each face of the object geometry, each triangle of the geometry, gets assigned three vertices from each `face` array such that the face is sliced into two triangles. The two triangles of the texture are then rendered onto the geometry. This is done for each triangle of the object geometry, therefore for the cube it is done twelve times. The vertices must be chosen carefully as if they are not chosen correctly the texture will be mapped incorrectly as well, as seen in Figure 5. Figure 1 displays correct mapping of the texture onto the cube geometry.
Given more time I would be interested to explore UV mapping applied to a spherical object, as that seems to be a good way to apply textures to a sphere.

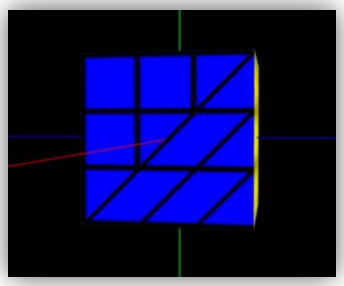

Figure 5: Incorrectly chosen vertices for the mapping cause visual artefacts.

Requirement 8 – Load a mesh model

Most of the difficulty of loading a mesh to be rendered is hidden away within the `OBJLoader` class. An object of that class is created and the `load()` function is used to load an object of a specified name, and run a function on load to process that object straight away. The "b" key on the keyboard is used to load and remove the mesh.

To load an object, the `OBJLoader` goes through every line of the object file and does something else depending on the first letter of the line. If the first letter is a "v", then the next three floats are assumed to be coordinates of a vertex and

are treated as such. If the first letter is a "f", then the next three numbers are lines of the object file where vertices are located that make up the face described. The face is created by drawing edges between all three vertices [12].

Once the mesh is fully loaded the `onLoad` function is run [13].

```
var boundingBox = new THREE.Box3().expandByObject(obj);
var boundingBoxSize = boundingBox.getSize();
var scaleX = cube.geometry.parameters.width / boundingBoxSize.getComponent(0);
var scaleY = cube.geometry.parameters.height / boundingBoxSize.getComponent(1);
var scaleZ = cube.geometry.parameters.depth / boundingBoxSize.getComponent(2);
```

First a bounding box is created and expanded by the object loaded in. The box is now the minimum bounding box of the object loaded in. This allows me to easily find out the size of the object, which is obtained via the function `getSize()`. Then the scale of the object respective to the cube is calculated for each dimension of the world space.

```
bunnyMatrix = new THREE.Matrix4();
bunnyMatrix.set(
      finalScale, 0, 0, -boundingBox.getCenter().x/2,
      0, finalScale, 0, -boundingBox.getCenter().y/2,
      0, 0, finalScale, -boundingBox.getCenter().z/2,
      0, 0, 0, 1
);
```

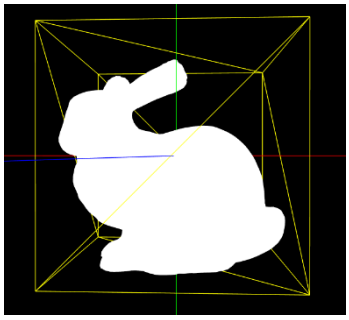

Figure 6: Correctly scaled and translated bunny.

In order to uniformly scale the object, the smallest scale out of the three is chosen and set as the `finalScale` variable. The object might be off centre, it might not be placed inside the cube correctly, therefore the object is set to be translated by half of the position of the centre of the bounding box.

This is all used to create a transform matrix called `bunnyMatrix` that the loaded object is then multiplied by in order to be scaled and translated correctly to fit inside the cube. The cube is replaced by a wireframe of itself so that the object can be seen without obstructions. The end result can be seen in Figure 6.

The transformation matrix above is a combination of a translation matrix and a scaling matrix in homogenous coordinates [14]. The matrix is structured as follows:

$$T = \begin{bmatrix} x & 0 & 0 & X \\ 0 & y & 0 & Y \\ 0 & 0 & z & Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where (x, y, z) are scale factors in each dimension, and (X, Y, Z) are translations in each respective axis.
Given more time I would like to explore how multiple objects could be loaded in and could interact with each other.

Requirement 9 – Rotate the mesh, render it in different modes

Rotating and rendering the loaded object has been done in almost the same way as with the cube. The same keys are used to rotate the mesh and render it in different modes as are used for the cube.

Loading in the object toggles the `bunnyMode` Boolean. If this value is true, then the key presses do different things. In section 3 of this report the code snippet can be seen to have that functionality, and it rotates the bunny using the same principles as rotating the cube. The rotation is done the same for every axis of the object.

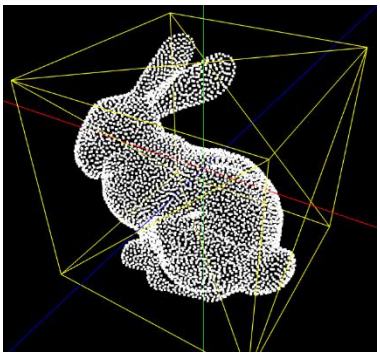

Figure 7: Vertex mode of the bunny object.

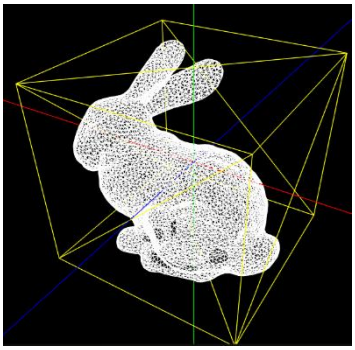

Figure 7: Edge mode of the bunny object.

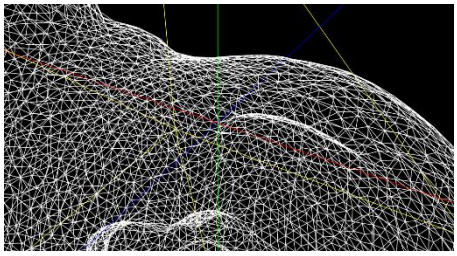

Figure 9: A close up of the edge mode of the bunny object. The triangles can be seen more clearly than in Figure 8.

The rendering in different modes is also done in the same way. For vertex mode (Figure 7), the object material is changed to a `PointsMaterial` with the points being set to be small (`0.05`) so that they do not overlap when rendered. Then a `Points` object is created from the old geometry and the `PointsMaterial`. For edge mode (Figure 8 and 9), a new material is created with the `wireframe` parameter set to `true`, this shows the edges of each face of the object. Face mode creates a `MeshPhongMaterial` and applies it to the bunny geometry, this can be seen in Figure 6.

A problem that I have found is that the bunny object does not rotate around the world origin, it rotates around its own centre which is not at the world origin. The centre of rotation of the bunny is equal to the translation vector ((X, Y, Z) from the transformation matrix discussed in section 8). This vector is equal to $[-0.5, 0, 0]^T$, so the centre of rotation of the bunny is that position on the world coordinates. This causes the bunny to rotate out of the cube when rotating in the x axis.

Given more time I would have liked to find out why is that the case, and how could I make sure that any object that is loaded in rotates around the world origin and not around another point in the world coordinates.

Requirement 10 – Let's get creative!

As a creative implementation I decided to explore how I could add sounds to the operation of the cube and object. The cube and loaded object each have their own theme. I have also implemented a so called "super mode" where the lighting and music change, and the rotation of the cube and object are increased. The background music is on always, and super mode can be triggered by pressing the "s" key. All the audio sounds are different themes from Super Mario Bros. namely the Overworld, Underwater, and Star theme, and also the Power Up and Power Down sounds. They have been obtained from YouTube.

For the sounds to work correctly three new objects from the three.js API are needed: `AudioListener`, `AudioLoader`, and `Audio`. The `AudioListener` object is created and attached to the camera. It is a global object that uses the Web Audio API to listen to any sounds that are being played and transfer them to the output [15, 16]. *NB: The Web Audio API is not supported by Internet Explorer. [16]*

```
var audioLoader = new THREE.AudioLoader();
audioLoader.load('Overworld.mp3', function(buffer) {
	overworld.setBuffer(buffer);
	overworld.setLoop(true);
	overworld.play();
});
```

The `AudioLoader` object can load audio buffers and assign them to `Audio` objects. The `Audio` objects themselves hold the buffer and can play, stop, pause, and do other things with the audio buffer. In the code snippet above, the `AudioLoader` object loads in the sound file called "Overworld.mp3" and assigns it to the `overworld` variable. The latter variable is then set to loop, and is played straight away using the `play()` function as it is the theme for when the cube is showing. There are a few more audio files loaded, but that is done in the same way.

Super mode changes the music played to the Star theme. It also changes the light type, and speeds up the rotation. Pressing "s" toggles the `superMode` Boolean. When this Boolean is true, the `AmbientLight` is replaced by a `PointLight`, and the `PointLight` changes colour and position every second. Time is counted using the Clock object available in three.js.

```
if(superMode){
oldTime = clock.oldTime;
	if(clock.getDelta() > 1){
		var randPos = Math.floor(Math.random() * 6);
		pointLight.position.set(
			pointLightLocations[randPos].x,
			pointLightLocations[randPos].y,
			pointLightLocations[randPos].z
		);
		pointLight.color.set(Math.random() * 0xffffff);
	} else {
		clock.oldTime = oldTime;
	}
}
```

The location of the `PointLight` is chosen randomly from a predetermined array of six possibilities. A random number is generated using the `Math` library and multiplied by six, as the `Math.random()` function generates a random float between zero and one [17]. The same is done to change the colour of the light, but the random float is multiplied by the hexadecimal representation of the colour white.

The time is checked by the `getDelta()` function of the Clock object. The start time of the clock, `oldTime`, has to be stored as it is reset whenever `getDelta()` is called [18]. If the time elapsed is below one second, then the start time of the clock is set as the `oldTime` variable. If it above one second, the position and colour of the light is changed.

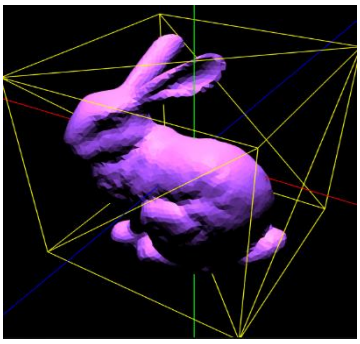

Figure 10: Bunny as seen in Super Mode.

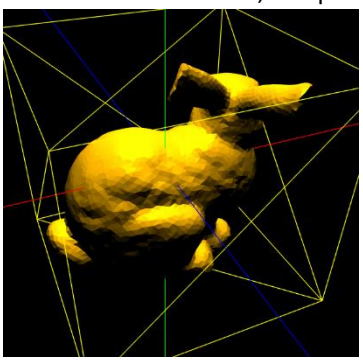

Figure 8: A different angle of the bunny in Super Mode.

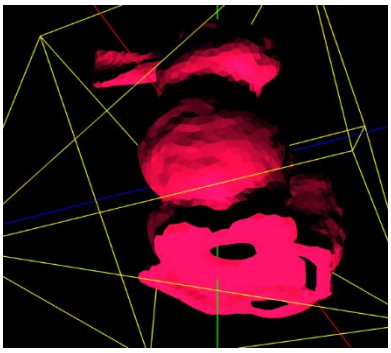

Figure 9: Bunny in Super Mode as seen from the bottom.

Figures 10 to 12 show the bunny lit up from different sides by different colours of light. PointLight is a directional light therefore can create shadows unlike AmbientLight. The bunny is a MeshPhongMaterial hence it gets shaded in using the Phong shading algorithm when hit by a light with a direction.

Given more time I would have liked to explore the audio capabilities of three.js and the Web Audio API to create audio visualizers, or a web based synthesizer or sampler. I would also have liked to explore more ways of shading objects, or shading different objects.

Conclusion

Overall, I have learned a lot during the development of above features. I have found it interesting to explore WebGL and three.js however the documentation for three.js is often quite unclear with regards to the functionality of certain objects, functions, etc. Nonetheless almost every requirement could be explored in more detail and expanded upon to get the best results and more interesting implementations.

References

1. https://threejs.org/docs/
2. https://threejs.org/docs/#api/materials/LineBasicMaterial, accessed 12th December 2017
3. https://threejs.org/docs/#api/objects/Line, accessed 12th December 2017
4. https://threejs.org/docs/#api/core/Object3D, accessed 13th December 2017
5. https://threejs.org/docs/#api/math/Quaternion, accessed 13th December 2017
6. https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles, accessed 13th December 2017
7. https://threejs.org/docs/#api/lights/AmbientLight, accessed 14th December 2017
8. https://en.wikipedia.org/wiki/Spherical_coordinate_system, accessed 14th December 2017
9. https://github.com/mrdoob/three.js/blob/master/src/math/Vector3.js, accessed 14th December 2017
10. https://en.wikipedia.org/wiki/Inverse_trigonometric_function, accessed 14th December 2017
11. https://en.wikipedia.org/wiki/UV_mapping, accessed 14th December 2017
12. https://github.com/mrdoob/three.js/blob/master/examples/js/loaders/OBJLoader.js, accessed 14th December 2017
13. https://threejs.org/docs/#examples/loaders/OBJLoader, accessed 14th December 2017
14. http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/#transformation-matrices, accessed 14th December 2017
15. https://threejs.org/docs/#api/audio/AudioListener, accessed 14th December 2017
16. https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API, accessed 14th December 2017
17. https://www.w3schools.com/jsref/jsref_random.asp, accessed 14th December 2017
18. https://threejs.org/docs/#api/core/Clock, accessed 14th December 2017