CM20219 – Viewing and analysing 3D Models using WebGL

Kieran Warwick

December 15, 2017

Introduction

In this coursework, the task is to implement software to render and manipulate 3D models using WebGL. This can be done by using the Three.js JavaScript graphics API created, by Cabello et al. (2010), which enables easy creation and display of 3D graphics in a web browser. In this report, we will analyse how to do tasks, such as creating and manipulating objects, then show how they are implemented through the Three.js API.

1 Draw a simple cube

The task is draw a cube centred at the origin with opposite corner points at $(1, 1, 1)$ and $(-1, -1, -1)$ with its faces orthogonal to the x, y, and z axes. This can be achieved by generating a geometry of a cube that the task requires, then creating a mesh with that geometry and a material. The mesh is then added to the scene. To implement this, the following code was used:

```
var geometry = new THREE.BoxGeometry(2, 2, 2);
var material = new THREE.MeshBasicMaterial({color: 0xffff00});
var cube = new THREE.Mesh(geometry, material);
scene.add(cube);
```

The BoxGeometry function creates a geometry of a box of width, height and depth of two. This defines the vertices and then the triangular faces of the cube by using the index of the vertices. The function MeshBasicMaterial creates a material to be able to draw the geometry. The mesh function constructs a 3D object out of the material and the geometry and by default the center position is $(0, 0, 0)$. A cube is then added to the scene.

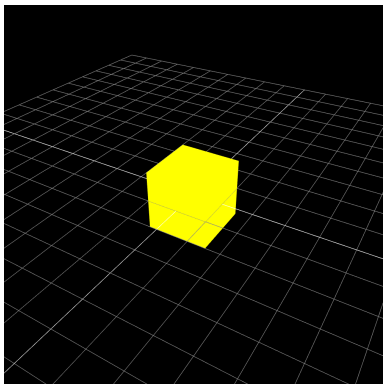


Figure 1: The cube as drawn on screen

To check that the cube is correctly drawn, it is necessary can look at the cube's geometry and look at the vertices. This is done with this simple line of code:

```
console.log(cube.geometry.vertices);
```

This prints the array of vertices that make the cube to the console. This array contains $(1, 1, 1)$ and $(-1, -1, -1)$ so we can conclude that the cube is drawn correctly.

To check the cube is in the right position, simply use this command:

```
console.log(cube.position);
```

It will then return a Three.js Vector3D representation of the position of the cube's center in the form (x, y, z). Doing this returns the vector $(0, 0, 0)$ so we can confirm that the cube center is correct. It is clear when drawn, as seen in figure 1

2 Draw coordinate system axes

The task is to draw orthogonal lines to represent the x, y and z axes of the world coordinate system. This can be achieved by using a AxesHelper which is found in the Three.js API. The implementation of this was simple:

```
var axes = new THREE.AxesHelper(500);
scene.add(axes);
```

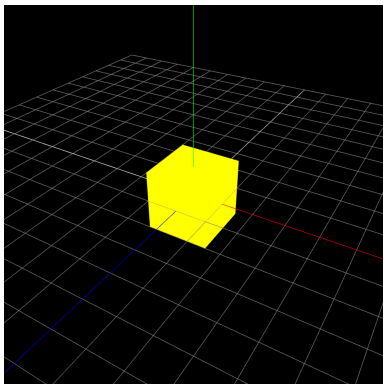


Figure 2: The axes on screen

The AxesHelper function generates 3 lines, each of length 500 with RGB colour for the x, y and z axes respectively. In the first instance, the lines were drawn individually, then the AxesHelper function was used since it facilitated better practice. Figure 2 clearly shows the the lines are correctly coloured and that the lines are drawn in the positive axes

3 Rotate the cube

The task is to rotate the cube about the x, y, and z axes without rotating the global axes or the camera. This means a rotation matrix needs to be applied to the cubes matrix.

Initially an intrinsic Euler rotation was used, $R = Z(\psi)Y(\phi)X(\theta)$, of order XYZ, where,

$$Z(\psi) = \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix}, Y(\phi) = \begin{pmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{pmatrix}, X(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix}$$

This can be simplified into one matrix

$$Y(\phi)X(\theta) = \begin{pmatrix} \cos(\phi) & \sin(\phi)\sin(\theta) & \sin(\phi)\cos(\theta) \\ 0 & \cos(\theta) & -\sin(\theta) \\ -\sin(\phi) & \cos(\phi)\sin(\theta) & \cos(\phi)\cos(\theta) \end{pmatrix}$$

$$Z(\psi)Y(\phi)X(\theta) = \begin{pmatrix} \cos(\psi)\cos(\phi) & \cos(\psi)\sin(\phi)\sin(\theta) - \cos(\theta)\sin(\psi) & \sin(\psi)\sin(\theta) + \cos(\psi)\sin(\phi)\cos(\theta) \\ \cos(\phi)\sin(\psi) & \cos(\psi)\cos(\theta) + \sin(\theta)\sin(\phi)\sin(\psi) & \cos(\theta)\sin(\psi)\sin(\phi) - \cos(\psi)\sin(\theta) \\ -\sin(\phi) & \cos(\phi)\sin(\theta) & \cos(\phi)\cos(\theta) \end{pmatrix}$$

However there was an issue: the gimbal lock phenomenon. When $\phi = \frac{3\pi}{2}$ this matrix gets produced:

$$Z(\psi)Y(\frac{3\pi}{2})X(\theta) = \begin{pmatrix} 0 & -\cos(\psi)\sin(\theta) - \cos(\theta)\sin(\psi) & \sin(\psi)\sin(\theta) - \cos(\psi)\cos(\theta) \\ 0 & \cos(\psi)\cos(\theta) - \sin(\theta)\sin(\psi) & -\cos(\theta)\sin(\psi) - \cos(\psi)\sin(\theta) \\ 1 & 0 & 0 \end{pmatrix}$$

$$Z(\psi)Y(\frac{3\pi}{2})X(\theta) = \begin{pmatrix} 0 & -\sin(\theta + \psi) & \cos(\theta + \psi) \\ 0 & \cos(\theta + \psi) & -\sin(\psi + \theta) \\ 1 & 0 & 0 \end{pmatrix}$$

This result implies that if the values of θ and ψ change in this matrix, it will have the same effect. This is what a gimbal lock is.

Instead quaternions were used, which are of the form:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} \sin(\frac{\theta}{2})e_x \\ \sin(\frac{\theta}{2})e_y \\ \sin(\frac{\theta}{2})e_z \\ \cos(\frac{\theta}{2}) \end{bmatrix}$$

where e is a vector and θ is the angle to rotate.
We can use these to rotate our cube around the axes by multiplying the cube's quaternion with a new quaternion. This avoids gimbal lock because there is no multiplication order, only a vector that is rotated around, thus no gimbal lock.

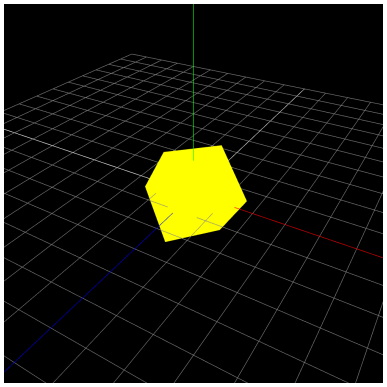


Figure 3: Showing the rotation of the cube

The following functions are needed to implement this:

```
cube.rotateX(0.05);
cube.rotateY(0.05);
cube.rotateZ(0.05);
```

The Three.js API provides a function to rotate the cube using the x, y, and z axes as the vector, only an angle to the function is needed. This function then creates the quaternion and multiples it to the objects quaternion. This is inside the animate loop so it will look like the cube is spinning at a constant rate. Global flags were used to determine if the rotation should be applied and flags set to toggle when the x, y, or z keys are pressed respectively. There is also a reset key-bind to the r key, which sets all the rotations back to 0 and the global flags to false.

To test that it is working, the key-binds are used to see if it spins in the correct axes. Testing confirms that it does. Although hard to see but in figure 3, all the axes have been rotated around.

4 Different render modes

The task is to render the cube using only vertices, edges or faces by assigning key-binds for each mode. This can achieved by making a new material, then removing the cube and applying the new material to the cube. The cube is then added again to the scene.

In order for the faces to be seen correctly, the lighting needs to be a directional light. This is because it will make the faces a different shade of yellow.

```
var directionalLight = new THREE.DirectionalLight(0xffffff, 0.5);
directionalLight.position.set(5,5,5);
```

This creates a directional light at the position $(5, 5, 5)$ which simulates sun light. This makes it much clearer to see the faces on the cube.

```
cube = new THREE.Mesh(cube.geometry, texture);
```

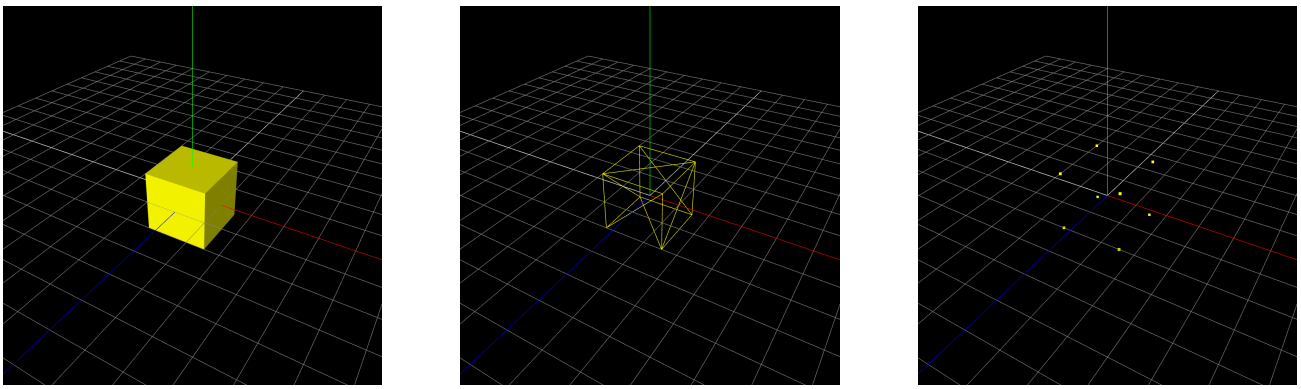
This creates a new cube based on the cube's original geometry and adds a material that renders the faces. The variable texture is an array of MeshPhongMaterial objects. This means that light will affect the cube so we can see the faces. The result can be seen in figure 4a and determined that it works as intended.

```
cube = new THREE.Mesh(cube.geometry, new THREE.MeshBasicMaterial({color: 0xffff00, wireframe: true}));
```

This creates a new cube with the original geometry, but changes the material property so it shows the wire-frame, which shows an outline of the all the primitive triangles that make up the cube. The results seen in figure 4b concludes it works.

```
cube = new THREE.Points(cube.geometry, new THREE.PointsMaterial({color: 0xffff00, size: 0.05}));
```

This creates a new Points object by using the original geometry and a new points material. A different material is used because points are handled differently to faces. A special material is required when using points.The result can be seen in figure 4c where it shows all the vertices of the cube, as required.



(a) Face render mode (b) Edge render mode (c) Vertices render mode

Figure 4: showing the diffrent render modes

5 Translate the camera

The task is to translate the camera's location along its relative x, y, and z axes. To do this, both the look at point and the camera are translated to keep the radius of the orbit controls constant. This is done by first using the camera's local axes which is retrieved from the cameras matrix. The local axis vector is then multiplied by a distance to calculate a vector which, when added to the camera's position vector, will move the camera along the local axis by the specified distance. The same vector is then used and added to the look at point position vector, in order to move it in the same way as the camera. This is implemented using the following code:

```
// Get the cameras local axes from the cameras matrix
var xAxis = new THREE.Vector3().setFromMatrixColumn(camera.matrix, 0);
var yAxis = new THREE.Vector3().setFromMatrixColumn(camera.matrix, 1);

// Calulate the change in postion
xAxis.multiplyScalar(xdist*0.05);
yAxis.multiplyScalar(ydist*0.05);

// add the change in postion to the target and the camera
target.add(yAxis).add(xAxis);
camera.position.add(yAxis).add(xAxis);
```

To translate the z axis however we only use this line of code:

```
camera.translateZ(event.deltaY*0.01);
```

The in-built function is the same except it does not edit the look at point's position. This means that the radius orbit of the orbit controls can be changed.

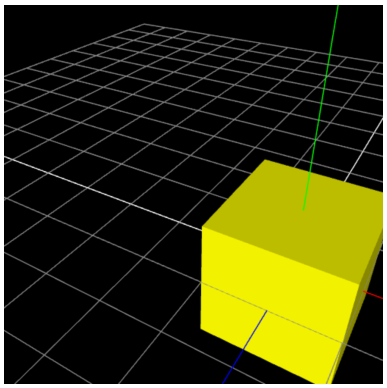


Figure 5: Translation of the camera

The mouse was set up to translate the camera by using event listeners. To translate the camera's x or y axes, right click then drag, and in order to translate in the local z axis, the scroll wheel is used. This creates an interactive system and make it extremely usable. Proof that the camera translates can be seen in figure 5.

6 Orbit the camera

The task is to manipulate the camera by rotating it about a look at point. This can be done by using the spherical coordinate system. Firstly convert Cartesian coordinate system into the spherical coordinate system from the by finding the radius r, polar angle ϕ and equator angle θ.

$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\phi = \arccos\frac{y}{r}$$

$$\theta = \arctan\frac{x}{z}$$

Then change the polar and equator angles to orbit about a point. Change ϕ to change the latitude (where $0 < \phi < \pi$) and change θ to change the longitude, and then convert back to into Cartesian coordinate system by finding

$$x = r\ \sin\phi\ \sin\theta$$

$$y = r\ \cos\phi$$

$$z = r\ \sin\phi\ \cos\theta$$

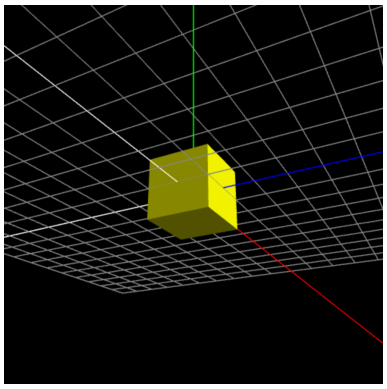


Figure 6: Showing orbit controls

This will get the new position to set the camera's new position. To implement this, the following code was used:

```
var spherical = new THREE.Spherical();

// The relative vector from the target to camera
spherical.setFromVector3(camera.position.sub(target));
spherical.theta += xdist*0.002; // Longitude
spherical.phi -= ydist*0.002; // Latitude

// Restrict phi to 0 < phi < pi
spherical.makeSafe();

// Set new camera position by changing the relative vector into a position vector
camera.position.setFromSpherical(spherical).add(target);
camera.lookAt(target);
```

Firstly, this finds the vector between the cube and the look at point. It then converts that vector into a spherical coordinate. Then it changes the longitude and latitude angles. The latitude angle is restricted to $0 < \phi < \pi$ in order to have unique coordinates. It is then converted back into a Cartesian coordinate and added to the position of the look at target to get a position vector. The cameras position is then set to this vector.

By using the same event listeners as used previously in the task to translate the camera. To get the angle to change the longitude and latitude angles by, we use the distance the mouse is moved while left click dragging. Proof that the radius is constant can be checked by the line of code:

```
console.log(camera.position.clone().sub(target).length());
```

This finds the radius of the camera around the look at target. The reason why the camera's position is cloned because it would affect the cameras position, which is not is wanted since the camera will be moved and will affect the result.

The result is that the camera is always at a constant distance when rotating around a point, as required. This is shown in figure 6.

7 Texture mapping

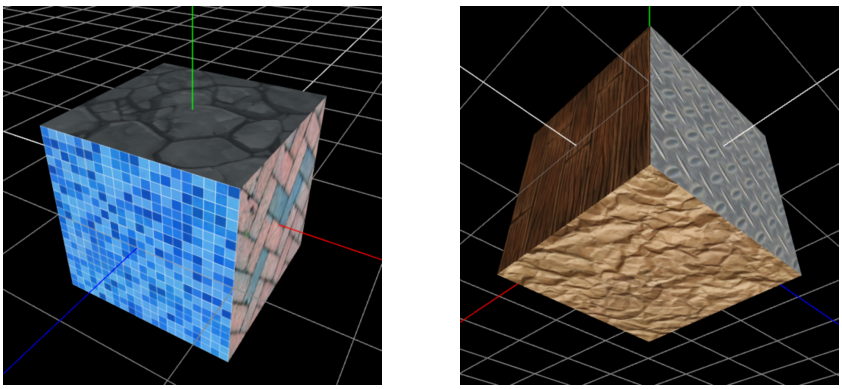
The task is to map different textures on different faces of the cube. This can be achieve by loading in images and mapping them to a material, then applying the material to the geometry. When an array of 6 materials is passed to the mesh function with a cube geometry, it will apply a unique material to each side of the cube. In order to implement, the following code was used:

```
var loader = new THREE.TextureLoader().setPath('images/'),
var texture = [ new THREE.MeshPhongMaterial({map:loader.load('texture0.jpg')}),}),
                new THREE.MeshPhongMaterial({map:loader.load('texture0.jpg')}),),
                ...
                new THREE.MeshPhongMaterial({map:loader.load('texture0.jpg')})];

cube = new THREE.Mesh(geometry, texture);
```

Using a texture loader to convert an image file into a texture object, the texture of the material is set to the image that was loaded in. An array containing materials is created where, each of the materials have been mapped with a unique image. Then this can be meshed into an object and then added to the scene.

The front side of the cube (as seen in figure 7a) and the back side of the cube (as seen in figure 7b) are viewed to test that it works, which it does.



(a) Front view of the cube (b) Back view of the cube

Figure 7: The diffrent textures mapped on the cube

8 Load a mesh model from .obj

The task is to load a Stanford bunny mesh, and then to translate and scale it uniformly to fit inside the cube. This is done by loading the .obj file using the OBJLoader.js library, created by Cabello et al. (2017), then scaled down by using the scale factor k,

$$k = \frac{\text{minimum length of the cube}}{\text{maximum length of the bunny}}$$

and finally translated by a offset vector Δ, where

$$\Delta = \text{cube position} - \text{center of the bunny}$$

The implementation is as follows:

```
var loader = new THREE.OBJLoader();
loader.load('bunny-5000.obj',
  function (obj) {
    bunny = obj.children[0];

    // scale the bunny down uniformly inside the cube
    ...
    var scale = minBound/maxBound;
    bunny.geometry.scale(scale,scale,scale);

    // translate to the point (0,0,0) from intial offset.
    ...
    bunny.geometry.translate(-offset.x,-offset.y,-offset.z)

    bunny.material = new THREE.MeshPhongMaterial({color: 0xffffff});
    scene.add(bunny);
  }
);
```

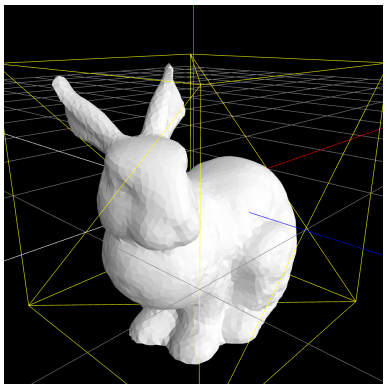


Figure 8: Showing the bunny

The reason why the children[0] is used for the bunny is because it is where the mesh is stored, otherwise it is a group object which would make it harder to manipulate.

To test that the bunny will will scale down for all sizes of cubes correctly, the size of the cube was changed to half the size to test if the bunny was still uniformly scaled and translated correctly. From the results, as seen in figure 8, it is clear that the bunny is scaled down uniformly and was translated inside the smaller cube.

9 Rotate the mesh, render it in different modes

The task is to perform for the bunny, the same render modes and rotations that the cube can perform. This allows reuse of code, so using the exact same code as for the cube, it will perform the task flawlessly. This means the implementation is very similar, for rotations:

```
bunny.rotateX(0.05);
bunny.rotateY(0.05);
bunny.rotateZ(0.05);
```

you can simplify this by making a toggle to control what object is being manipulated which was assigned to the space-bar. This means that it is possible to do the following:

```
var obj = toggleBunny ? bunny : cube;
obj.rotateX(0.05);
obj.rotateY(0.05);
obj.rotateZ(0.05);
```

This simplified the code down so it more portable, but for the render modes, it is not possible to do this. To show the faces, as seen in figure 9a, the following code is used:

```
bunny = new THREE.Mesh(bunny.geometry, new THREE.MeshPhongMaterial());
```

For the edge render mode, as seen in figure 9b, the following code is used:

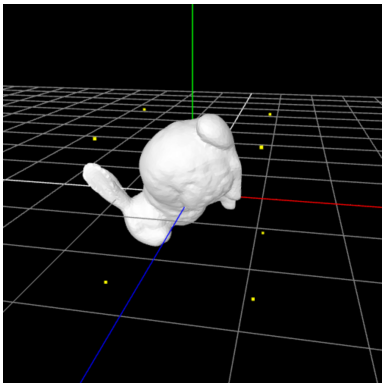
```
bunny = new THREE.Mesh(bunny.geometry, new THREE.MeshBasicMaterial({wireframe: true}));
```

Then for the vertices render mode, as you can see in figure 9c, was implemented the following way:

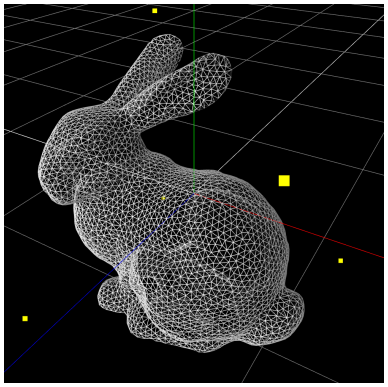
```
bunny = new THREE.Points(bunny.geometry, new THREE.PointsMaterial({size: 0.005}));
```

Unfortunately, this creates a lot of repeated code, which could be reduced down into functions that take in an object and do render modes, but because of how I implemented the render modes, it does not work as expected. It fails to remove the bunny or cubes. I did find a workaround but it made the code unreadable so I choose not to put it in.

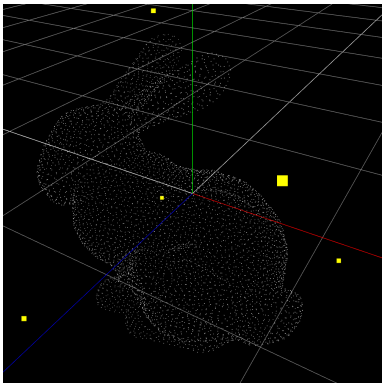
Testing that the tasks are performed can be clearly seen in figure 9



(a) Face render mode and rotation



(b) Edge render mode



(c) Vertices render mode

Figure 9: showing the diffrent render modes

10 Be creative – do something cool!

10.1 Creating a skybox

This was fairly simple, all was needed was six images to show in the positive and negative axes. This is implemented by:

```
var loader = new THREE.CubeTextureLoader().setPath('images/');
scene.background = loader.load([
  'px.jpg', 'nx.jpg',
  'py.jpg', 'ny.jpg',
  'pz.jpg', 'nz.jpg'
]);
```

which makes a texture cube and sets the scene's background to it. The result can be seen in figure 10.

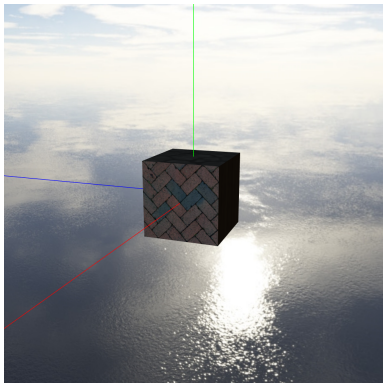


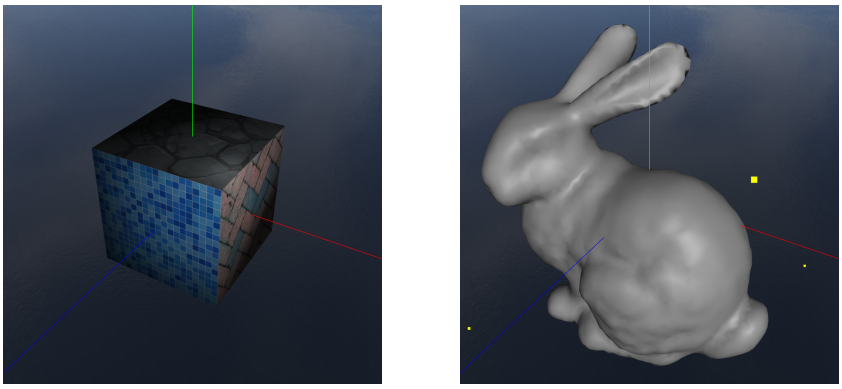
Figure 10: Showing the skybox

10.2 More realistic shading

In order to have realistic shading, the material used was a MeshPhongMaterial. This was because it calculates shading per pixel which gives more accurate results than other materials. The implementation is as follows:

```
geometry.computeVertexNormals();
geometry.computeFaceNormals();
```

This computes the face normals and the vertex normals, which gives Phong shading because the vertex normals are interpolated from those at each vertex of the face. The results can be seen in figure 11



(a) Cube's Phong shading (b) Bunny's Phong shading

Figure 11: Advanced shading for models

10.3 Shadows

Firstly, set the render to have a shadow map, so it will map the shadows onto objects, then set the light cast shadows and then define all objects that will cast a shadow. The implementation is this:

```
renderer.shadowMap.enabled = true;
directionalLight.castShadow = true;
plane.receiveShadow = true;
```

The result of this can be seen in figure 12

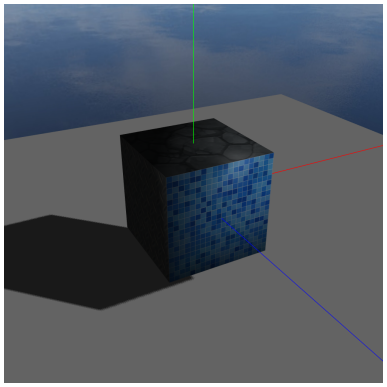


Figure 12: Showing the shadow of the cube

10.4 Orbits around a star

The planets are created using a sphere geometry and a Phong material. The speed of the planet is then calculated using Newton's law of universal gravitation, the formulae for the speed is

$$v = \sqrt{\frac{GM}{r}}$$

where G is the gravitational constant, M is the mass of the star and r is the radius of the orbit.

```
var speed = Math.sqrt(0.00000000006754*centerMass/orbitRadius);
```

To orbit a point, spherical coordinates were used and θ was changed. This below is the orbit function:

```
spherical.setFromVector3(obj.position);
spherical.radius = obj.userData.orbitRadius;
spherical.theta += obj.userData.speed;
obj.position.setFromSpherical(spherical);
obj.rotation.y += 0.05;
```

In the animate loop, orbit was called to make it appear that the planets are orbiting at a constant rate

```
for(var i=0; i<planets.length; i++){
  orbit(planets[i]);
}
```

A orbit outline was also added to show the orbits the planets take, this is done by using a CircleGeometry of the radius of the orbits, removing the first vertex, then mesh into a LineLoop object with a LineDashedMaterial.

Moons were added to some the planets and they are stationary relative to the planet. Then by rotating the planet in the y axis, the moons appear as they are orbiting the planet. Shadows were also added.

The result of all this can be seen in figure 13.

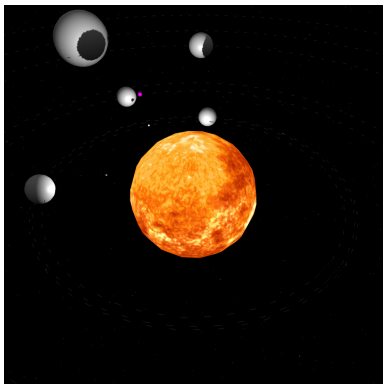


Figure 13: Showing the orbiting planets

10.5 Procedural galaxy

Using all the features above, the creation of a galaxy seemed easy, but there was issues with hierarchy, as the planets did not move as intended. To fix this issue, the code was rewrote more generally. When done, it was realised that making it procedural would be as simple as just inputting different parameters to functions, such as number of planets and the mass of the star. So putting random variables in most parameters, it become a procedural galaxy.

All the planets were stored in an array so each one can orbit its star. It looked strange with the orbits in the same direction so there was added a 50 % chance of a reverse orbit. Then when zooming out, then it looked weird that all the orbits where just flat. So a tilt object was added to add all the planets too. This made it possible to tilt the orbits by editing the rotation of the tilt object.

Shadows and realistic shading were added as well, the same way as described previously. A starry background was made using points and random numbers.
The implementation of the procedural galaxy is shown below:

```
generateSystem(0xffffff, 10000000*Math.random()+10000000, new THREE.Vector3(Math.random()*500 -250,Math.
    random()*500 - 250,Math.random()*500 -250), 10*Math.random()+5);
function generateSystem(colour, centerMass, position, numberOfPlanets){
  ...
  for(var i=2; i<numberOfPlanets+2; i++){
    generatePlanet(colour, Math.random()*i*i + 10, stars[stars.length -1]);
  }
}

function generatePlanet(colour, orbitRadius, star){
  ...
  var tilt = new THREE.Object3D();
  tilt.position.copy(star.position);
  tilt.rotation.x = star.userData.tilt[0];
  tilt.rotation.y = star.userData.tilt[1];
  tilt.rotation.z = star.userData.tilt[2];
  tilt.add(planets[planets.length - 1]);
  ...
  if(Math.random()<0.5){
     generateMoon(0xff00ff, planets[planets.length - 1], 10);
  }
}
```

The result of this can be seen in figure 14

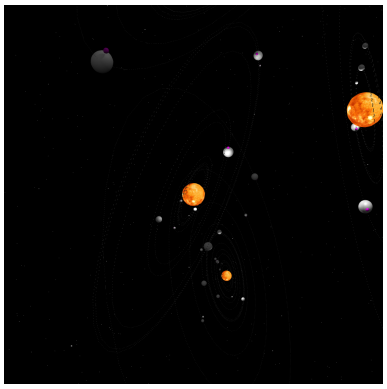


Figure 14: Showing the procedural galaxy

Conclusion

The creation and manipulations of objects in a Three.js environment is simple and it is easy to do complex things, such as rotations without gimbal lock or shadows, with a function in the Three.js library. The limitations of the library that I encountered was the lack of pointers in JavaScript language so I could not simplify some functions. Other than that, it is a great way to learn how to view and analyse 3D models. It also hides a lot of complex functions of WebGl into simple functions, which is a brilliant thing to do.

References

Cabello, R. et al., 2010. *Three.js* [Online]. Available from: `https://github.com/mrdoob/three.js` [Accessed December 15, 2017].

Cabello, R. et al., 2017. *Objloader.js* [Online]. Available from: `https://github.com/mrdoob/three.js/blob/master/examples/js/loaders/OBJLoader.js` [Accessed December 15, 2017].