Viewing and analysing 3D models using WebGL

Introduction

In this paper, a simple webpage is constructed and used to render and manipulate different 3D models using WebGL. The basic structure behind the program is that an HTML file loads in a Javascript, which includes all of the Three JS libraries. Three JS in turn wraps WebGL, hence it allows writing WebGL application in JavaScript on any given webpage. This report attempts to establish the background theory and mathematics, as well as implementing key elements from a given requirement specification. Source code will be included in small sections where relevant.

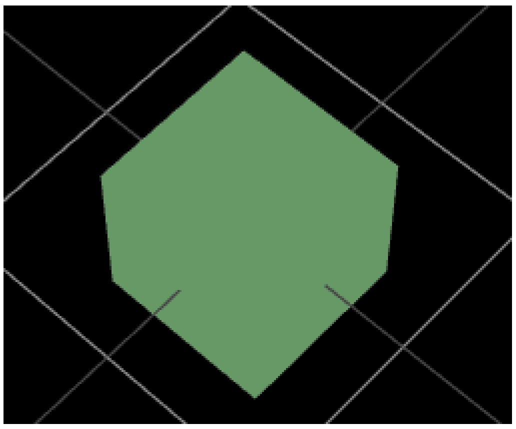
Assignment

Requirement 1: Draw a simple cube

This requirement served as an introduction to using three.js, the premise of it was simple: make a cube, put it in the centre, add it to the scene. This was done as follows:

```
//Setup the size of the starting cube
var boxX = 1;var boxY = 1;var boxZ = 1;
//Setup the shape of the cube (a BoxGeometry)
var geometry = new THREE.BoxGeometry(boxX, boxY, boxZ);
//Set the material (wrapping texture) to be a light green)
var material = new THREE.MeshBasicMaterial({ color: 0x7777ff});
//Merge them into the cube Mesh
var cube = new THREE.Mesh(geometry,material);
//Add the mesh to scene
scene.add(cube);
```

With the result being visible in figure 1:



This is a simple introduction to using WebGL and three.js. Later it is covered why MeshBasicMaterial might not be the best choice due to shading issues and similar. The cube is centred at origin (0,0,0) with opposite corner points (-1,-1,-1) & (1,1,1). Faces are orthogonal to the x-, y-, and z- axes. No real difficulties were encountered here.

Figure 1 1x1x1 Cube

Requirement 2: Draw coordinate system axes

The objective here is to visualise a three-dimensional axes system using three orthogonal lines in red, green and blue, to make it easier to see X, Y and Z further on. This was achieved as follows:

```
// Create a visualization of the axes of the global
// coordinate system (requirement 2)
var axesHelper = new THREE.AxesHelper(5, 1);
//Offset the axes to be slightly higher than the ground grid,
// to prevent them from clipping into the grid
axesHelper.position.set(0.001,0,0.001);
// Add the Axeshelper to the scene
scene.add(axesHelper);
```

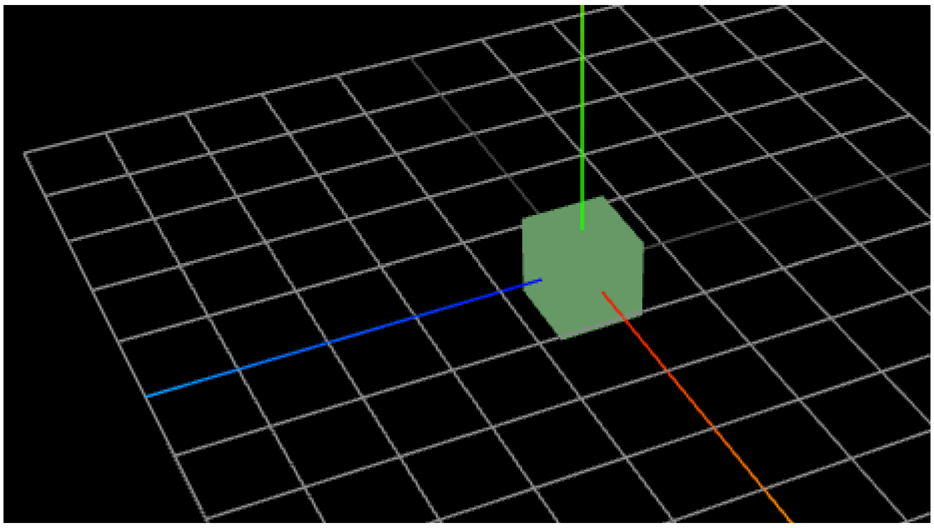


Figure 2 XYZ indicator axes

In figure 2 it is clear there was achieved the 3 orthogonal lines, stemming out of the centre and going to the end of the grid. The only difficulty encountered with this was the fact that the grid tended to overlap the axes – this was fixed by offsetting the axes position by a negligible amount, so that the system knows that they are to be displayed on top, without significantly altering their position as to cause it to be in an erroneous place in the plane.

Requirement 3: Rotate the cube

The premise of this requirement is to be able to press a button and have the cube rotate. Initially opting to hold the button down and have the rotation increment each time, but instead put a variable that allowed it to repeat infinitely until the button was next pressed. Achieved as follows:

```
// Handle keyboard presses.
function handleKeyDown(event){
    switch (event.key){
        //if the corresponding button is pressed, start its rotation
        case 'x':if(rotateX == 0){rotateX = 1;}else{rotateX = 0;}break;
        case 'y':if(rotateY == 0){rotateY = 1;}else{rotateY = 0;}break;
        case 'z':if(rotateZ == 0){rotateZ = 1;}else{rotateZ = 0;}break;
    }
}
```

So when x, y or z is pressed, a variable ,rotateXYZ, will flip from 0 to 1 or 1 to 0. In the animation part the following is put:

```
// Each animation rotation, check if rotateX is set,
// If it is set, rotate it by a slight amount (rotateSpeed)
if(rotateX == 1){cube.rotation.x += rotateSpeed;}
if(rotateY == 1){cube.rotation.y += rotateSpeed;}
if(rotateZ == 1){cube.rotation.z += rotateSpeed;}
```

In figure 3 there is an example of rotating around the X-Axis. The program allows you to rotate in all directions, as well as reset it by pressing R, which simply resets all cube.rotation.x/y/z back to 0.

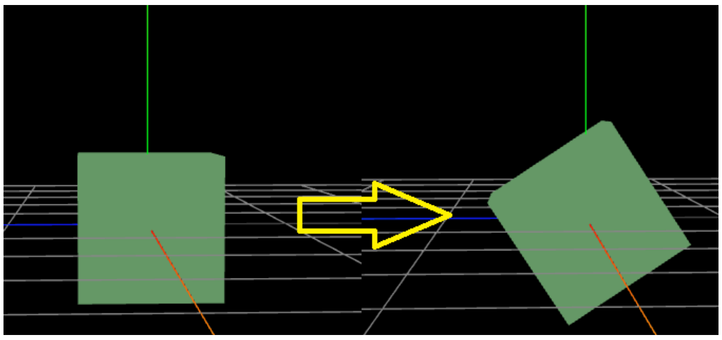


Figure 3 Rotating the cube around the X axis

Requirement 4: Different Render Modes

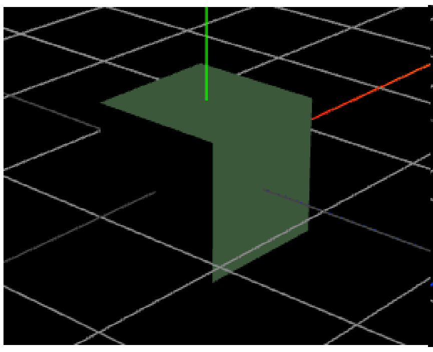


Figure 4 Faces

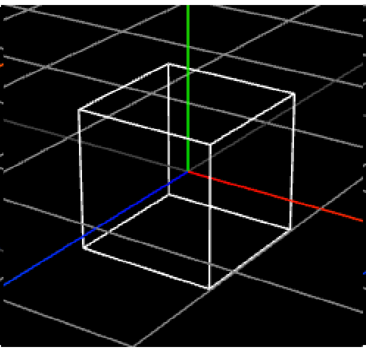


Figure 4 Edges

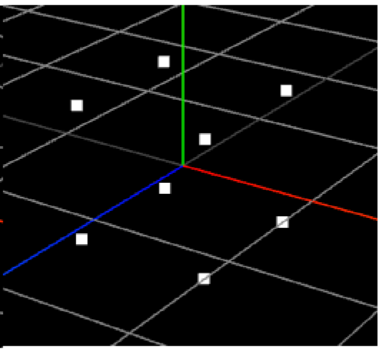


Figure 6 Vertices

The fourth requirement was required to explore the the different rendering modes, showing the 6 faces of the cube (3 at a time) seen in Figure 4, the edges of the primitives (the "corners" or vertices) seen in Figure 5, and the edges seen in figure 6, in some cases known as a "wireframe".

For the faces, the material for the cube was simply changed from a BasicMaterial to LampertMaterial:

```
material = new THREE.MeshLambertMaterial( {color: 0x669966} );
```

And casting some light onto it as to ensure the difference in Faces is visible:

```
//Add some light
light = new THREE.DirectionalLight(0xffffff);
light.position.set(10, 10, 10);
light.target.position.set(3, 0, 0);
light.castShadow = true;
scene.add(light);
```

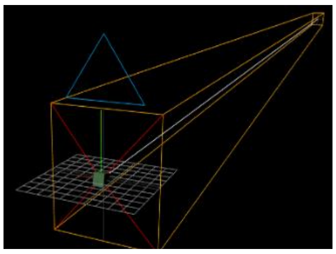


Figure 5 The light debug frame

With the results of this being visible in Figure 4. There was a phase to add a debug button to ensure it was known where the light was coming from and how it functioned. By pressing D the light source can be viewed.
This is due to the following "helper":

```
helper = new THREE.CameraHelper( light.shadow.camera );
```

The D button simply adds or removes it to the scene, as seen in Figure 7.

For edges, create a function found in three.js called EdgesGeometry of our geometry, which finds the edgelines and creates an object out of them, after which we create edgeLines, which simply takes the edges object and draws them out using LineBasicMaterial. The result of this is seen in Figure 5; by simply adding edgeLines to the scene.

```
var edges = new THREE.EdgesGeometry(geometry);
var edgeLines = new THREE.LineSegments(edges,
                new THREE.LineBasicMaterial({color: 0xffffff}));
```

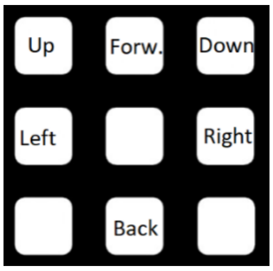
Vertices was done in a similar manner, but in reverse. Rather than creating the geometry first, we create the material first – by making vertexMats a single point of size 8, then using the three.js library THREE.Points to detect the points in the given object and display them using the materials given by vertexMat. Adding vertices to the scene gives the result seen in figure 6.

```
var vertexMats = new THREE.PointsMaterial({ size: 8,
                          sizeAttenuation: false, transparent: true });
var vertices = new THREE.Points (geometry, vertexMats);
```

Requirement 5: Camera Translation

This requirement consisted of translating the camera. The key to this requirement was to translate the camera along its own left, right, etc. vectors, rather than the axes of the global coordinate system. At first one could easily think of it as simply being an exercise of moving the camera position an X position left, right, etc. but this would use the global coordinate system. Instead three.js comes with the built-in translateOnAxis(axis,distance); function for the camera. This made it very easy, and originally the idea was built for requirement 6 off using this combined with repeating to look at the centre, but more on this later.

```
case '4':camera.translateOnAxis(xAxis, -0.05);break;
case '6':camera.translateOnAxis(xAxis,  0.05);break;
case '8':camera.translateOnAxis(zAxis, -0.05);break;
case '2':camera.translateOnAxis(zAxis,  0.05);break;
case '7':camera.translateOnAxis(yAxis,  0.05);break;
case '9':camera.translateOnAxis(yAxis, -0.05);break;
```



As the camera is moved in 3d (including forward/backward and up/down) using the arrow keys didn't feel very intuitive – as you would have 4 arrow keys plus up/down in a different place, hence there was opted to place it on the numerical keys, as can be seen on figure 8. This would later allow forming rudimentary orbital controls by centring the camera after moving it, but this was quickly replaced with a much better mathematical function, more on this in the next section, covering requirement 6.

Figure 8 Keypad Layout

Requirement 6: Orbiting the camera.

For requirement 6 the requirement was to build orbital controls – much like the pre-built package OrbitalControls. In essence this is moving the camera around the object in such a way that it is as if the camera were attached to a sphere, spinning around a centrepoint, much like can be seen in Figure 9, initially there was the consideration of the reusing the camera translation, but this ended up not working very well, looking very choppy unless you did small iterations multiple times – it simply wasn't very intuitive. After this there was opted for more of a mathematical approach!

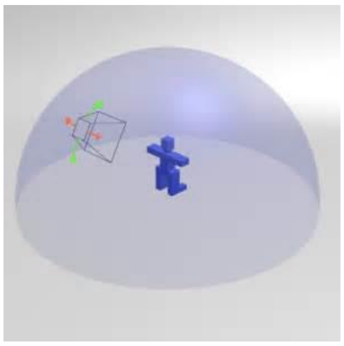


Figure 9 A rudimentary drawing for spherical camera control

Initially the setup required a few things, as a start, a listener for the mouse movement, as well as mousewheel (for zooming in and out):

```
            document.addEventListener( 'mousemove', onDocumentMouseMove);
            document.addEventListener( 'mousedown', onDocumentMouseDown);
            document.addEventListener( 'mouseup', onDocumentMouseUp);
            document.addEventListener( 'mousewheel', onDocumentMouseWheel);
```

Now, on to polar coordinates. For all other parts of this the Carthesian coordinates were used (referring to x, y and z), primary because it is a simpler method to understand visually in integer amounts, rather than Polar coordinates (or Spherical), as seen in figure 10

The spherical coordinates use the radius, as well as the two angles theta and phi to determine its current location. One can keep the radius at the exact same position and change phi and theta. Now, we've already covered how the radius adjusts the zooming distance, phi and theta are handled as follows:

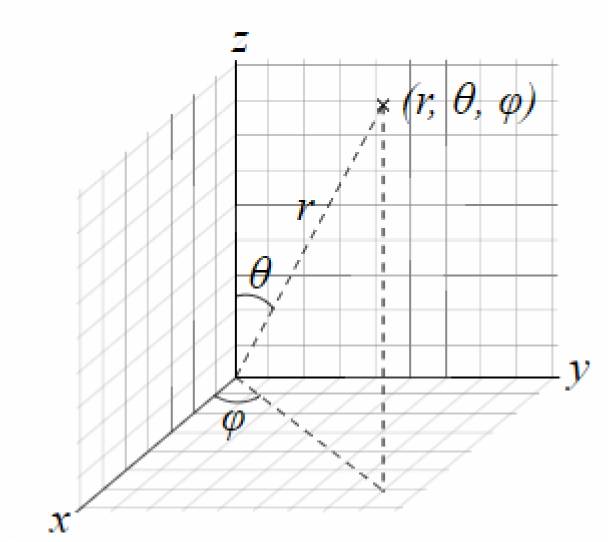


Figure 10 Spherical Coordinates in an XYZ 3D system

The following two formula form the basis of finding phi and theta:

$$\theta = \arccos\left(\frac{r}{z}\right)$$
$$\varphi = \arctan\left(\frac{y}{z}\right)$$

Then, using the formulae that turn Spherical back into Cartesian coordinates, we can reverse the operation as to make it easier to translate either way.

$$x = r * \sin(\theta)\cos(\varphi)$$
$$y = r * \sin(\theta)\sin(\varphi)$$
$$z = r * \cos(\theta)$$

Now to implement these into translating the camera, we first used the listeners as to where the mouse was held down (onMouseDownX), then where it was moved to

```
function onDocumentMouseDown(e){
    mouseIsUp = false;
    mouseIsDown = true;
    onMouseDownTheta = theta;
    onMouseDownPhi = phi;
    onMouseDownX = event.clientX;
    onMouseDownY = event.clientY;
    }
}
```

:

```
function onDocumentMouseMove( e ) {
    if ( mouseIsDown ) {
        theta = - ( ( e.clientX - onMouseDownX ) * 0.5 )+ onMouseDownTheta;
        phi = ( ( e.clientY - onMouseDownY  ) * 0.5 )+ onMouseDownPhi;
        camera.position.x = camZoomSet*(Math.sin(theta * Math.PI / 360 )
                            * Math.cos( phi * Math.PI / 360 ));
        camera.position.y = camZoomSet*Math.sin(phi*Math.PI / 360 );
        camera.position.z = camZoomSet*Math.cos(theta* Math.PI / 360 )
                            * Math.cos( phi * Math.PI / 360 );
    }
    camera.lookAt(centre);
}
```

From this you can see that the original location of the mousedown is used by subtracting the current position from it and translating it into polar coordinates as it moves, edging the camera to its destination. The camera can be sped up and made more sensitive, but the 0.5 here is hardcoded.

After the spherical coordination was added, implementing a mousewheel zoom was an easy task. Originally there was opted to use the wheelDeltaY, but instead simply cheking whether it was going up or down with positive/negative worked just as well – if not better.

This was achieved as follows:

```
function onDocumentMouseWheel(e){
        if (e.wheelDeltaY > 0){      //scroll up
        camZoomSet -= 0.25;
        }else{                       //scroll down
        camZoomSet += 0.25;
        }
}
```

camZoomSet is a variable that is used to basically multiply the polar coordinates' radius – as adjusting the amplitude of this would simply change the proximity to the centre of the "sphere".

Requirement 7 : Texture Mapping

The texture mapping required essentially 6 different faces to be put onto the cube, this was achieved by using the built-in TextureLoader and addings these onto Meshes on each side, as the cube material is made up of an array of sides, this can be broken into 6 different sides, each forming a part of the cube. Adding this to the scene looked as is seen in Figure 11, with the code behind it being:

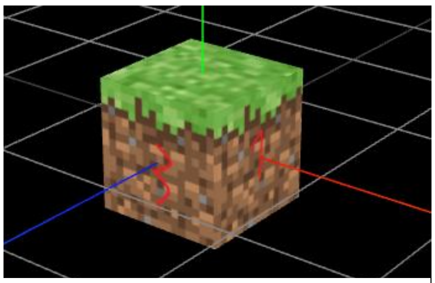


Figure 11 Textured cube

```
var loader = new THREE.TextureLoader();
textureMaterial = [
   new THREE.MeshBasicMaterial({map: loader.load('sides1.png')}),
   new THREE.MeshBasicMaterial({map: loader.load('sides2.png')}),
   new THREE.MeshBasicMaterial({map: loader.load('top.png')}), //top
   new THREE.MeshBasicMaterial({map: loader.load('bottom.png')}), //bottom
   new THREE.MeshBasicMaterial({map: loader.load('sides3.png')}),
   new THREE.MeshBasicMaterial({map: loader.load('sides4.png')})
];
scene.remove(cube);
cube = new THREE.Mesh(geometry,textureMaterial);
scene.add(cube);
```

Requirement 8: Load a mesh model from .obj

For requirement 8, the model was required to be loaded in, a bunny. Using the three.js OBJLoader the bunny was loaded in and added to the scene using a single function application:

```
var loader = new THREE.OBJLoader();
loader.load('bunny-5000.obj',
function ( object ) {bunnyObject = object;
     scene.add ( bunnyObject );
     },
);
```

The second objective of the requirement was to size the bunny according to the cube. This was done by creating a box around the bunny that is the size of the bunny to determine its extremities, then settings its size to be equal to that of the cube. The positioning however simply takes 0.25 times the X axis, which would work for a bunny and box of any size.

The code for this is as follows:

```
function sizeBunny(){
    box = new THREE.Box3().setFromObject(bunnyObject);
    //Set the bunny length using the box position
    bunnyXlength = box.max.x - box.min.x;
    bunnyYlength = box.max.y - box.min.y;
    bunnyZlength = box.max.z - box.min.z;
    //Scale the bunny
    bunnyObject.scale.x = boxX / bunnyXlength;
    bunnyObject.scale.y = boxY / bunnyYlength;
    bunnyObject.scale.z = boxZ / bunnyZlength;
    //position it
    bunnyObject.position.x = -0.25*boxX;
}
```

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

Rotating the mesh was done in a similar fashion as rotating the cube – using a true/false Boolean that checked whether a button had been pressed, then simply rotated it.

```
if(bunnyButton){
    bunnyObject.rotation.x += rotateSpeed;
}
```

The requirement specification only specified to do it in one axis, but this can easily be expanded to multiple.

Different render modes for the bunny were done in a similar fashion again as the cube, as the geometry of the bunny can be found using its object properties children, it is simply plugging in the same edges and vertices code modified to use the bunny object's child property geometry instead of the cube.

```
bunnyVertices = new THREE.PointCloud(bunnyObject.children["0"].geometry,
vertexMats);
bunnyEdges = new THREE.EdgesGeometry(bunnyObject.children["0"].geometry);
bunnyEdgeLines = new THREE.LineSegments(bunnyEdges, new
THREE.LineBasicMaterial({color: 0xffffff}));
```

Requirement 10: Creativity!

For the tenth requirement we were given a *carte blanche* of essentially doing whatever we wished to do, as long as it was creative and intended to bring something new. I wanted to create a little interactive game – originally I had intended to make a Minecraft-esque voxel game, but I saw multiple other people had done this so opted for a more creative approach.

Instead I created a simple basketball game engine, it had a ball bouncing in 3d you could push forward, and you could watch it bounce across the screen – it didn't have any goals or scoreboard, in a way it was similar to a game of Squash. You see how many times you can hit the ball before it bounces back from the wall. The code for this implemented an interesting ray device, when you push P this "ray" becomes active, the ray is a vector that locates your mouse's polar coordinates and shoots a "beam" and waits for first intersection with any object – when it intersects with the ball, it has 6 cases (the ball is surrounded by an invisible 6-sided cube) as to where to push the ball. The ball also slowed down as time went on, the speed of bounce, height, length and walls (where the ball bounced back) were all adjustable, making it essentially a strong foundation of an engine for a simple basketball game. The most important part, the "vector ray" code is as follows:

```
function onDocumentMouseDown(e){
    vector = new THREE.Vector3( ( event.clientX / window.innerWidth ) * 2 -
1,
                        - ( event.clientY / window.innerHeight ) * 2 + 1,
0.5 );
   vector.unproject( camera );
   raycaster.set( camera.position,
vector.sub( camera.position ).normalize() );
   intersects = raycaster.intersectObject( ballCube );
   if ( intersects.length > 0 ) {
      index = Math.floor( intersects[0].faceIndex / 2 );
      switch (index) {
         case 0:
            XSpeed = 0.1; goXMinus = true; goXPlus = false;
            break;
         case 1:
            XSpeed = 0.1; goXPlus = true; goXMinus = false;
            break;
         case 2:
            if(bounceConstant <= 8){bounceConstant++;}
            break;
         case 3:
            if(bounceConstant >= 3){sphere.position.y = bounceConstant -
1;bounceConstant--;}
            break;
         case 4:
            ZSpeed = 0.1; goZMinus = true;goZPlus = false;
            break;
         case 5:
            ZSpeed = 0.1; goZPlus = true; goZMinus = false;
            break;
      }
}
```

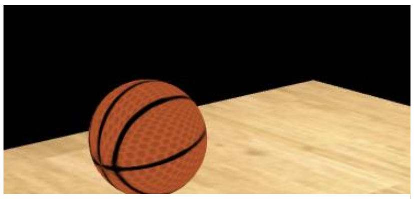


Figure 12 the Baskbetball game in action

On figure 12 the textured ball can be seen having freshly bounced off the (textured) floor. The focus of the mini-game is the usage of the ray to control the ball – the rest simply ties it together as a whole.

Conclusion

In conclusion, WebGL is quite a powerful item to have at hand – and simply runs in a browser window! It is easy to see the applications in both academia and leisure, as the basketball game is good proof of its applications in a more relaxed context. The application of the mathematics behind implementing 3D rendering and manipulations is also of particular interest, as this is the first time I've seen an actual application behind the morphing of Carthesian coordinates into polar coordinates – many years after learning what these are and how to use them. Applying theoretical knowledge into a practical, feasible application in a stepwise manner is a great way of learning.

References:
Ohner, A. (2015). Orbital Camera. [online] Available at https://andreasrohner.at/posts/Web%20Development/JavaScript/Simple-orbital-camera-controls-for-THREE-js/ [Accessed 01/12/17].
MIT (2010). Three js documentation [online] Available at https://threejs.org/ [Accessed 22/11/'17]