Viewing and analysing 3D models using WebGL

WebGL is a JavaScript programming interface for 2D and 3D graphics, compatible with many web browsers. The JavaScript library three.js is a programming interface for 3D graphics rendering and can be used without cumbersome plugin overheads due to WebGL's compatibility with modern web browsers. By exploring 3D rendering using three.js, a more comprehensive understanding of translation, rotation and lighting in 3D could be gained.

1.0 Draw a simple cube

The first task was to render a cube centred at the origin (0, 0, 0) with the right front corner vertex at (1, 1, 1) and the left front corner vertex at (-1, 1, 1). This was accomplished by using the THREE.BoxGeometry() constructor to specify the width, height and depth of the cube (Figure 1); using the THREE.MeshBasicMaterials() constructor to set a colour for each of the faces (Figure 1); and using the THREE.Mesh() constructor to combine the geometry and materials to create a cube as seen in Figure 2.

```
// Set geometry for cube.
cubeGeometry = new THREE.BoxGeometry(1,
1, 1);

// Set colour for cube faces.
var cubeFace = new
THREE.MeshBasicMaterial({ color: 0x62ff00
});

// Draw the Cube
cube = new THREE.Mesh(cubeGeometry,
cubeFace); scene.add(cube);
```

Figure 1 Code for creating cube.

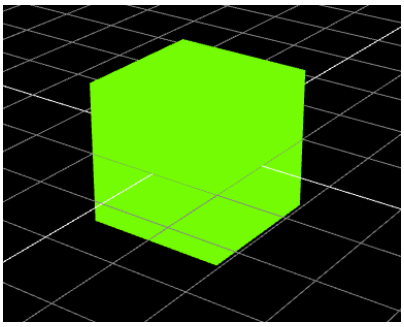

Figure 2 Cube Rendered at the origin

1.1 Background

THREE.BoxGeometry seen in Figure 1, is a geometry class in three.js that gives the user the option to specify width, height and depth. Geometries in three.js can store attributes such as vertices, colours, positions. THREE.BoxGeometry comes stored with attributes common to rectangular cuboids [2]. This cube, as the viewer sees it, only has three faces shown at any one time. This is because any more rendering would be wasted, since only three faces can physically be seen at any one time. Triangles are the polygon of choice in computer graphics, since a triangle is the most basic polygon, whose vertices all lie on the same plane. The cube drawn in Figure 1 would at a minimum be rendered using 6 triangles, two for each of the 3 faces seen.

2.0 Draw coordinate system axes

The second task to perform was drawing the coordinate system for the x, y and z-axes. This required the creation of three constructors using THREE.LineBasicMaterials() with different colours shown in Figure 3. Then three different geometries were created, each having vertex beginning at the origin and vertex a set distance in the direction of one of the axes shown below in Figure 4.

```
// Set colours for each axis
var xMat = new
THREE.LineBasicMaterial({
color: 0xfc0303});
var yMat = new
THREE.LineBasicMaterial({
color: 0x03fc07});
var zMat = new
THREE.LineBasicMaterial({
color: 0x0314fc});
```

Figure 3 Colours Red, Green and Blue assigned to x, y and z axes respectively

```
// Set the length and direction of each axis
by creating two vertices.
var xGeometry = new THREE.Geometry();
xGeometry.vertices.push(
new THREE.Vector3(0, 0, 0),
new THREE.Vector3(3, 0, 0));
var yGeometry = new THREE.Geometry();
yGeometry.vertices.push(
new THREE.Vector3(0, 0, 0),
new THREE.Vector3(0, 3, 0));
```

Figure 4 Length and direction of x and y axes assigned accordingly.

Figure 5 shows the axes drawn for x (red), y (green) and z (blue).

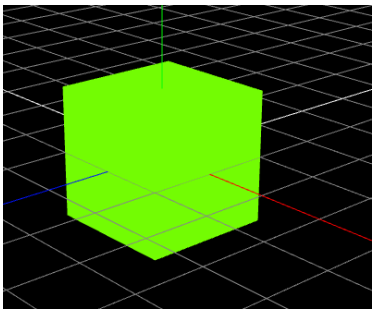

Figure 5 RGB axes drawn

3.0 Rotate the Cube

The third task was to enable the cube to rotate around each of the axes. By specifying each axis of rotation in the animate function block, shown in Figure 6 below, and incrementing each axis rotation by a given amount, the cube was made to rotate.

```
// Animation loop function. This function is
called whenever an update is required.
function animate() {
    requestAnimationFrame(animate);
    //rotate the cube faces
    cube.rotation.x += 0.01;
    cube.rotation.y += 0.01;
    cube.rotation.z += 0.01;
```

Figure 6 Animate function with x, y and z axis rotations of the cube.

This functionality was further extended to create a stop/start mechanism. For a demonstration point of view, this allows for a more streamlined demo. If all the rotations occur at the same time, it is impossible to see whether each axis rotation was accomplished without changing the parameters manually. Thus with the start/stop, the rotations do not have to be set to zero every time an individual rotation needed to be viewed. The start/stop mechanic was implemented by creating an event listening function, cubeRotate(), and using the keys 'x', 'y' and 'z' to toggle the axis rotations as seen in the code below (Figure 7). Creating an event listener has to do with what is known as Document Object Model (DOM), an HTML programming interface that represents the webpage so that programming languages can have access to the page. Using the addEventListener() method, an event listener can be appended to the DOM [3]. The addEventListener parameters take an event type, such as 'keydown' or 'mouseup', and a listener, which can be an Event Listener interface or in the case of this task, a JavaScript function.

Figure 7 shows the toggle feature for only the x rotation; the toggling of y and z axes operates in the same way, the only difference is in variable names and the keycode.

```
// Function that rotates the cube on key presses.
function cubeRotate(event) {
    switch (event.keyCode) {
        case 88: // x = start/stop x rotation.
            if (rot_X == 0.0) {rot_X = 0.01;
            }
            else {rot_X = 0.0;
            } break;
```

Figure 7 Code snippet for cube rotation of the x-axis in cubeRotate() function.

The global variable rot_X, shown above, replaced the number 0.01 in the animate function shown in Figure 6 for the x axis rotation. The global variables rot_Y and rot_Z do the same thing but for y and z rotations respectively.

3.1 Background

Each rotation of the cube uses a rotation matrix for the x, y and z axis shown in equation 1 below, where α, β and γ are in radians.

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix}, R_y(\beta) = \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix}, R_z(\gamma) = \begin{bmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (1)$$

The animate() function shown in Figure 6 uses the method requestAnimateFrame() with the callback "animate". The method requestAnimateFrame() tells the browser that an animation is about to be performed, and that the function in the callback should be updated before the next repaint [3]. This animate() function thus updates the amount of radians added to the Euler angle, shown in each matrix above, in the specified axis, 60 times per second (or whatever refresh rate is being utilised). Each vertex of the cube is multiplied by the new matrix values which achieves a real-time rotation.

4.0 Different render modes

Next, different render modes were added to the scene. A key press with the letter 'e' rendered the edges of the cube. Pressing 'v' rendered only the vertices of the cube and 'f' rendered the faces of the cube as shown in Figure 8.

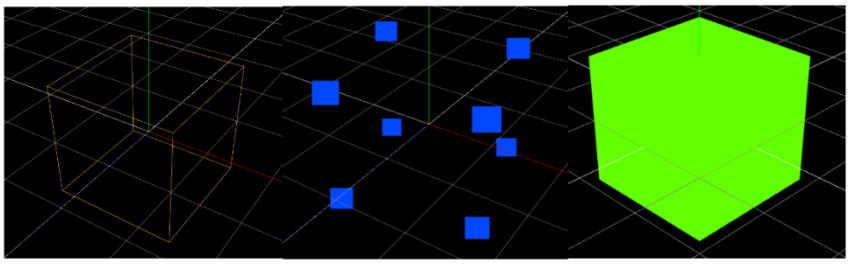

Figure 8 From left to right, edge, vertices and face rendering.

The code in Figure 9 shows added line and point geometries by using the already constructed cube geometry from 1.0. Figure 10 shows the code from the edge render mode where the other modes are removed from the scene and the edge rendering is added with the press of the 'e' key.

```
// Draw Edges, line is called in function
cubeRender.
var edges = new THREE.EdgesGeometry(cubeGeometry);
line = new THREE.LineSegments(edges, new
THREE.LineBasicMaterial({ color: 0xf5b342,
linewidth: 1 }));
// Draw Vertices, points is called in function
cubeRender.
var pMat = new THREE.PointsMaterial({ color:
0x0048ff, size: 0.4 });
points = new THREE.Points(cubeGeometry, pMat);
```

Figure 9 Edge and vertex geometry creation

```
// Function to render edges, faces and
vertices of the cube on keyboard presses.
function cubeRender(event) {
        switch (event.keyCode) {
            // Render modes.
            case 69: // e = edge
                scene.add(line);
                scene.remove(cube);
                scene.remove(points);
                break;
```

Figure 10 Function to render cube in different modes. Edge mode is shown.

Furthermore, each rendered mode was made to rotate at the same rate as the cube rotation. By including each variable for the edges(line) and vertices(points) in the animate() loop, each mode could be rotated at the same time as the cube faces (Figure 11).

```
//rotate the cube edges
line.rotation.x += rot_X;
line.rotation.y += rot_Y;
line.rotation.z += rot_Z;

//rotate the cube vertices
points.rotation.x += rot_X;
points.rotation.y += rot_Y;
points.rotation.z += rot_Z;
```

Figure 11 Code snippet of animate function with edge and vertex rotation added

4.1 Background

The vertices of the cube are stored in the geometry of the cube when it is first created, thus, by adding a polygon and a colour to the vertex coordinate, the vertices of the cube can be rendered. To draw edges of a cube, WebGL can use two vertices as the start and end point for each line. In this case the eight corners of the cube are used as the vertices, and lines are drawn between them for the edges of the cube. By placing a material into the mesh rendering constructor, THREE.Mesh(geometry, material), the faces of the cube are rendered with a colour, or a texture, if one is loaded into the material constructor.

5.0 Translate the Camera

Task five was accomplished by adding an event listening function called cameraTranslate(). When 'w', 'a', 's' or 'd' are pressed, the camera translated up on its y-axis, left on its x-axis, down on its y-axis or right on its x-axis respectively. The numbers 2 and 3 translated the camera backwards and forwards on its z-axis. Figure 12 shows the code snippets for translating the camera right and up on the camera's axis.

5.1 Background

Much the same way the rotations occur in task 3, a method for translating the camera was used to increment the camera

```
// Function that translates the camera on
key presses.
function cameraTranslate(event) {
switch (event.keyCode) {
        case 68: // d = right camera
        translation.
                camera.translateX(0.2);
                break;
        case 87: // w = up camera
        translation.
                camera.translateY(0.2);
                break;
```

Figure 12 Code snippet used to translate the camera on a key press.

location in space by the specified number of units and the specified axis. Translations are accomplished with a transformation matrix multiplication, shown in equation 2 below.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (2)$$

This matrix is multiplied to the column vector of the coordinates of each vertex of the cube to achieve the transformation. As each key press only increments in one axis, the transformation matrix only needs the specific transform entry for one axis in order to perform the translation, the other transformation entries are zero. Using the method camera.translateX translates the camera along its own x-axis. If instead, camera.position.x was used, the camera would move along the scene's x-axis.

6.0 Orbit the camera

The sixth task was to manipulate the camera so that it orbited around a "look at" point in an "arc ball" mode, that is, the camera would stay a set distance away from the look at point, but move longitudinally and latitudinally relative to the look at point in an "arc". This functionality could be implemented using key presses or mouse controls. Key presses were easier to implement, but this would mean more key bindings to set as well as a less common mode of implementation. Mouse controls are ubiquitous in computer generated camera viewing and so was concluded to be a worthwhile mechanic to implement. Figure 13 shows three different screen captures of the camera in different positions about the cube at a distance of 8 units.

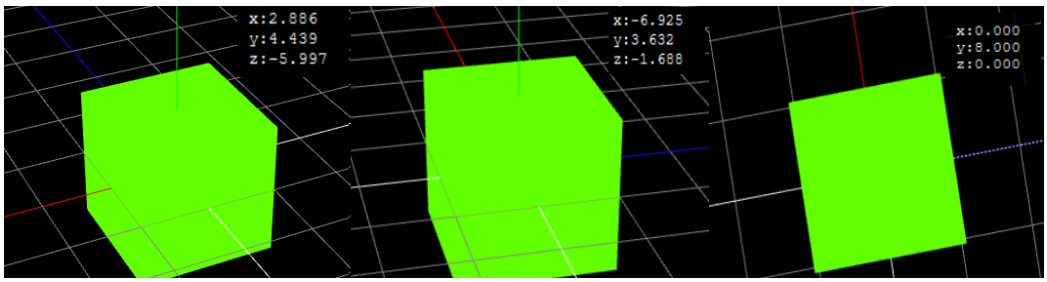

Figure 14 From left to right, looking down from negative z-axis, looking from negative x-axis looking directly down to the top of the cube

Global variables were created for the mouse controls and to give the camera its initial location as seen below in Figure 14 and Figure 15.

```
// Variables for orbiting camera with
mouse
var mousePosition, r = 8, azimuth =
45, mouseAzimuth = 45;
var inclination = 60, mouseInclination
= 60, isMouseDown = false;
```

Figure 13 Global variables associated with mouse controls

```
// Set up the camera, move it to (4, 4, 5)
camera = new THREE.PerspectiveCamera(45, window.innerWidth /
window.innerHeight, 0.1, 1000);
camera.position.x = r * Math.cos(azimuth * Math.PI / 180) *
Math.sin(inclination * Math.PI / 180);
camera.position.z = r * Math.sin(azimuth * Math.PI / 180) *
Math.sin(inclination * Math.PI / 180);
camera.position.y = r * Math.cos(inclination * Math.PI / 180);
```

Figure 15 Conversion from spherical to cartesian coordinates

The first mouse event listener sets the mouse position when it is pressed, and sets the isMouseDown flag to true. The code can be seen in Figure 16. For the second mouse event listener, the function checks if the down flag is true. If true, the function calculates the degree of movement of the mouse and places those data into inclination (longitude component) and azimuth (latitude component). Any movement that occurred to change the distance of the camera to the look at point is now entered to recalculate the radius of the "sphere" that the camera will orbit along, with the function radiusRecalc(). These new values of the radius, inclination and azimuth are then placed back into the equations seen in Figure 15 above to get the new position of the camera.

```
// Function to handle when the mouse is pressed.
function mouseClickDown(event) {            event.preventDefault();
        isMouseDown = true;// Sets down flag for mouse move function.
        mouseInclination = inclination;
        mouseAzimuth = azimuth;

        // Store the location of where the mouse is pressed.
        mousePosition.x = event.clientX;
        mousePosition.y = event.clientY;
```

Figure 16 Mouse Click function

Finally, the third mouse event listener is called when the mouse press is released, storing the location at which the mouse was released and setting the isMouseDown flag to false.

6.1 Background

Movement of the camera in an "arc ball" is done by calculating the spherical coordinates of the camera, changing the latitude and longitude angles with the mouse, and then recalculating the camera position back into cartesian coordinates. When calculating spherical coordinates, many texts assume that the z-axis is the axis pointing up, seen in Figure 17.

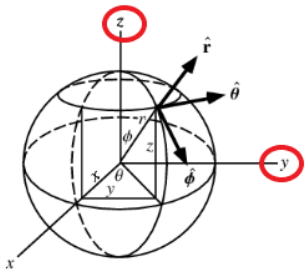

Figure 17 Spherical Coordinate diagram

This became a difficulty in testing with regards to establishing why the camera did not arc as the equations predicted, until the realisation that WebGL renders its scenes with the y-axis pointing up. Consequently, the y and z equations for calculating the camera position were swapped as seen in Figure 15. Additionally, when the user moves the mouse, an intuitive approach, for example, is to click and hold on the top of the screen, move the mouse to the bottom of the screen and the scene follows the movement of the mouse. It is for this reason that the inclination variable is inverted (Figure 18), to keep the longitudinal movement more intuitive to the users' inputs.

```
inclination = - ((event.clientY - mousePosition.y) * 0.1) + mouseInclination;
```

Figure 18 Inverted inclination for mouse control

7.0 Texture Mapping

The seventh task saw a different texture applied to each cube face, as well as different lighting and shadows. This was done by loading eight textures into one MeshPhongMaterial variable called faceMaterials, code shown in Figure 19.

```
// Textures for faces of the cube
var loader = new THREE.TextureLoader();
var faceMaterials = [
new THREE.MeshPhongMaterial({ map: loader.load('milfordSound.jpg') }),
new THREE.MeshPhongMaterial({ map: loader.load('reykjavik.jpg') }),
new THREE.MeshPhongMaterial({ map: loader.load('TanzaniaLion.jpg') }),
new THREE.MeshPhongMaterial({ map:
loader.load('churchofthegoodshepherd.jpg') }),
new THREE.MeshPhongMaterial({ map: loader.load('blue-lagoon.jpg') }),
new THREE.MeshPhongMaterial({ map: loader.load('Zanzibarcrooz.jpg') }),
];
```

Figure 19 Loader used for six face textures

Once these textures were applied, the cube was rendered as seen below in Figure 20 with ambient lighting and Figure 21 with point lighting.

Originally the lighting was ambient lighting, which in three.js does not allow shadows to be cast. This form of lighting was changed to point light source, set at the coordinates (5, 10, 10) with a white colour. With point light the parameter, castShadow can be set to true to allow objects in the scene to cast their shadow.

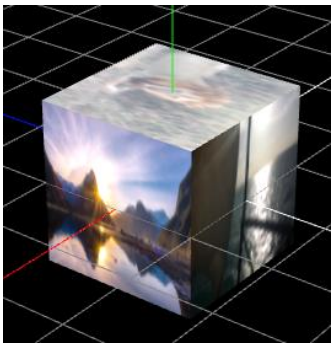

Figure 20 Textured cube with ambient lighting

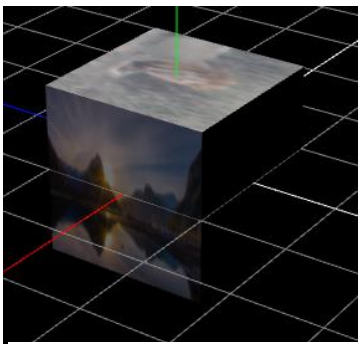

Figure 21 Textured cube with point lighting

7.1 Background

Every pixel that is drawn onto a scene has different values associated with it, such as the position of the camera, where the light source is, how strong the light source is and what the pixel's material is. All these criteria give different shader values to the pixels and from these values, different algorithms have been developed to mimic lighting found in the real world. The most popular of these is called the Phong lighting model. Phong modelling approximates real-world lighting by using ambient light, specular highlighting and diffuse highlighting [4]. This method replaced the interpolation of lighting between vertices by instead "approximating the shading point from the orientation of the approximated normal [4]". This Phong method is less computation heavy and instead focuses on getting a good approximation of the curvature of different surfaces for realistic shading.

8.0 Load a mesh model from .obj

The bunny-5000.obj was loaded into the scene. This was done with OBJLoader, which has as its parameters, the object's url and other optional parameters; namely, an option to callback the object just loaded. Utilising the callback allowed a variable to be set to the loaded object, and for its scale factor and position to be changed to fit inside the cube (Figure 22). The code that loaded, scaled and positioned the rabbit can be seen in Figure 23.

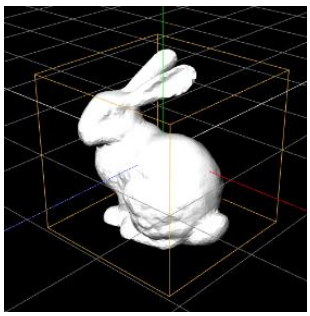

Figure 23 bunny-5000.obj positioned in the centre of the

```
//Load local object into scene.
var rabLoader = new THREE.OBJLoader();
rabLoader.load(
    // resource URL
    'bunny-5000.obj',
    // Callback to loaded object
    function (object) {
        mrBun = object;// Set global variable to
loaded object
        mrBun.scale.set(0.25, 0.25, 0.25); // Scale
bunny to fit  inside cube
        mrBun.position.set(-0.25, 0, -0.0625)
```

Figure 22 OBJLoader with scaling and translation

This was further extended to allow the object to be added or removed from the scene with a key press event listener seen below in Figure 24. Adding this functionality was important since the next task was to render different modes of the object. An ability to toggle the modes would need to be added in order to differentiate between them.

```
//Function to render edges, faces and vertices
of the rabbit on keyboard presses.
function rabbitRender(event) {
    switch (event.keyCode) {
        case 66:// b = add bunny to scene.
            scene.add(mrBun);
```

Figure 24 Key press to render object code snippet

8.1 Background

The scaling of any object in three-dimensional space makes use of the transformation matrix for scaling, shown in equation 3 below,

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (3)$$

where α, β and γ are scaling factors for x, y and z respectively. When scaling the bunny object, a uniform scaling was accomplished by scaling every vector within the object to one quarter the original position. To position the object centred in the cube, uniform translation was not possible since the centre of the object was not its most central position in space, shown in Figure 25.

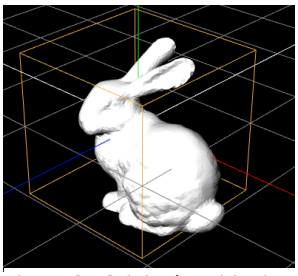

Figure 25 Original positioning of bunny-5000.obj without scaling

Instead, the object's position was translated in the x and z direction in different amounts to place the bunny in the centre of the cube.

9.0 Rotate and render the object in different modes

As with the cube, the loaded object was rendered with its edges, vertices and faces using the function rabbitRender() as a key press event listening function. To render the object's edges and vertices, however, was very different from the cube. The cube has 8 vertices while the bunny object has 5000. In order to access each vertex, draw them and the lines between them, a traverse() method was used on the object's stored geometries. At every child element that was a mesh class, a vertex could be drawn the same way as with the cube using THREE.PointsMaterial and THREE.Points. The edges were drawn in much the same way, except with the added variable using THREE.EdgesGeometry. Each edge was drawn with THREE.LineSegments included with a THREE.LineBasicMaterial. To change the colour of the faces of the rabbit, the child of the object needed to be traversed in the same way to update every face with the new colour. Figure 26 shows the different render modes of the loaded bunny object.

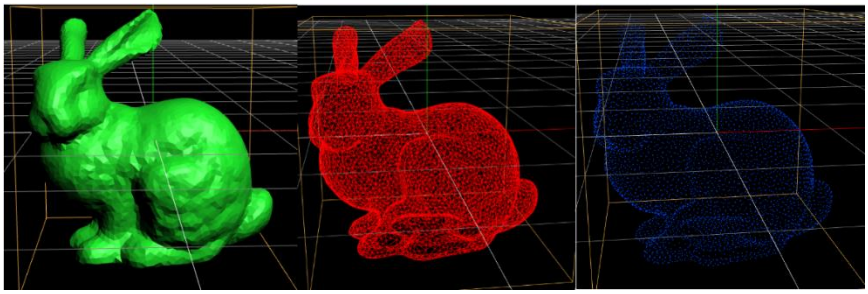

Figure 26 bunny-5000.obj render modes: faces, edges and vertices

Rotating the object took more testing to correct when compared to the cube. When a global variable storing the bunny object was placed in the animate() loop, a TypeError was thrown saying that the global variable was undefined (Figure 27).

Figure 27 Loaded object is undefined

This error, however, did not occur when placed in a different function, for example the rabbitRender() function. The clue here was that the rabbitRender() function was not called until a certain key was pressed by the user, ie it was only called after time had elapsed since calling the object load function. The animate() function uses the requestAnimateFrame() method as explained above in section 3.1. This method starts callback on the animate() function as soon as the scene is rendered, thus, if the object has not loaded at the same time the scene is rendered, the variable is undefined. Therefore, rotating the bunny object used flags initialised to false/zero, and then set to true only when a key press occured to account for the time it takes the scene to load the object.

10.0 Extending the Task

The first part of the extension was to place the scene into a background that could be viewed 360 degrees. This would further enhance the implemented mouse orbit controls from task 6.0. After some research, it was found that this 360 degree scene was called a skybox. A skybox uses a cube map set of pictures to create a cube enclosing a scene, ie. the scene is now inside of a cube with the textures rendered on the inside of each face. Three.js has a loader function to place each of the 6 images into a cubemap in the order pos-x, neg-x, pos-y, neg-y, pos-z, neg-z. The scene was placed inside a winter neighbourhood cube map downloaded from Emil Persson, aka Humus at http://www.humus.name [1] as shown in Figure 28.

Figure 28 Scene inside cubemap

A form of voxel painting was also added, where the voxels were made to be cones with a Christmas Tree texture applied to the outside, see Figure 29 below.

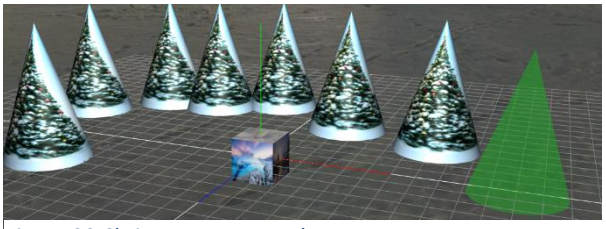

Figure 29 Christmas Tree Voxels

The Christmas tree painting coding consisted of, creating an invisible plane to place the trees, creating a transparent tree that moved with the mouse to assist in placement of the trees, and creating a solid tree with the texture applied that could be placed. The tree painting was able to be toggled on and off with the 't' key which set a flag called treeOn. Ray-casting was used in implementing this tree "painting" functionality. Ray-casting is essentially casting a ray from the mouse pointer through the frustrum of the scene and computing where the mouse intersects any objects [2]. Figure 30 below shows the code for ray casting with the roll over tree.

```
// Calculates position of the mouse. The coords are normalised in the device
mouse.set((event.clientX / window.innerWidth) * 2 - 1, - (event.clientY / window.innerHeight) * 2 + 1);
// Creates a raycaster object that starts from the mouse coords above, and uses the direction vector of the camera to cast the
ray into the scene.
raycast.setFromCamera(mouse, camera);

var intersects = raycast.intersectObjects(objects);

if ((intersects.length > 0) && (treeOn == true)) {
    var intersect = intersects[0];
    rollOverTree.position.copy(intersect.point).add(intersect.face.normal);// Renders the rollover tree at intersections to
the paint plane
    rollOverTree.position.divideScalar(.8).floor().multiplyScalar(.85).addScalar(.4);// The positioning of the rollover tree
```

Figure 30 Raycasting for rollover tree

This ray-casting above was also applied to the solid Christmas tree. The 't' key toggles the painting functionality on or off. When painting was toggled on and the 'p' key was held, a solid Christmas tree was placed on the paint plane with every mouse click. Likewise, the solid trees could be removed if the 'shift' key was held down and the mouse clicked on the selected tree.

Conclusions

By being able to work in a 3D rendering environment, a better understanding of how objects are rendered and how they can be manipulated was gained. Some important factors when dealing in 3D space include: knowledge of how the axes are displayed in the working environment is important when calculating which axis to rotate about; ray casting uses the camera's direction to paint a ray through to the scene to manipulate objects with the mouse; and maths behind rotations and translations may not give an intuitive mechanic to user interfaces without manipulating the equations used for the transforms. Future work with three.js could include implementing physics with the objects rendered, such as gravity and colliding with other objects in the scene.

References

[1] E. Persson, "http://www.humus.name/," 11 December 2019. [Online]. Available: http://www.humus.name/index.php?page=Textures&start=112. [Accessed 4 December 2019].

[2] "three.js – JavaScript 3D library," 28 November 2019. [Online]. Available: https://threejs.org/. [Accessed 6 12 2019].

[3] "MDN Web Docs," 6 December 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame. [Accessed 6 December 2019].

[4] B. T. Phong, "Illumination for Computer Generated Pictures," *Communications of the ACM,* vol. 18, no. 6, pp. 311-317, 1975.