CM20219 - CW 2 - 2020/21

December 18, 2020

1 Draw a simple cube

The cube was drawn with side dimensions of 2. Since the default for *scene.add* is to centre the cube at the origin $(0, 0, 0)$, side lengths of 2 gives the corner coordinates $(-1, -1, -1)$ and $(1, 1, 1)$. the *wireframe* parameter simply makes the cube easier to see, and the *color* is blue.

```
//Basic Cube
var boxGeo = new THREE.BoxGeometry(2,2,2);
var boxMat = new THREE.MeshBasicMaterial({color: 0xffffff, wireframe: true});
var cube = new THREE.Mesh(boxGeo,boxMat);
scene.add(cube);
```

Figure 1

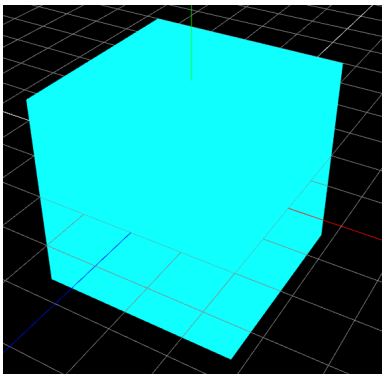

Figure 2

2 Draw coordinate system axes

The axes are simply drawn using the built-in *AxesHelper* function.

```
var axesHelper = new THREE.AxesHelper(20);
scene.add(axesHelper);
```

Figure 3

3 Rotate the cube

Rotation of the cube is performed by adjusting the object (cube) *rotation* property. the *rotateCube*() function is called in *animate*() so that it is constantly updated.

```
function rotateCube(){
        cube.rotation.x += 0.01;
        cube.rotation.y -= 0.01;
        cube.rotation.z -= 0.01;
}
```

Figure 4

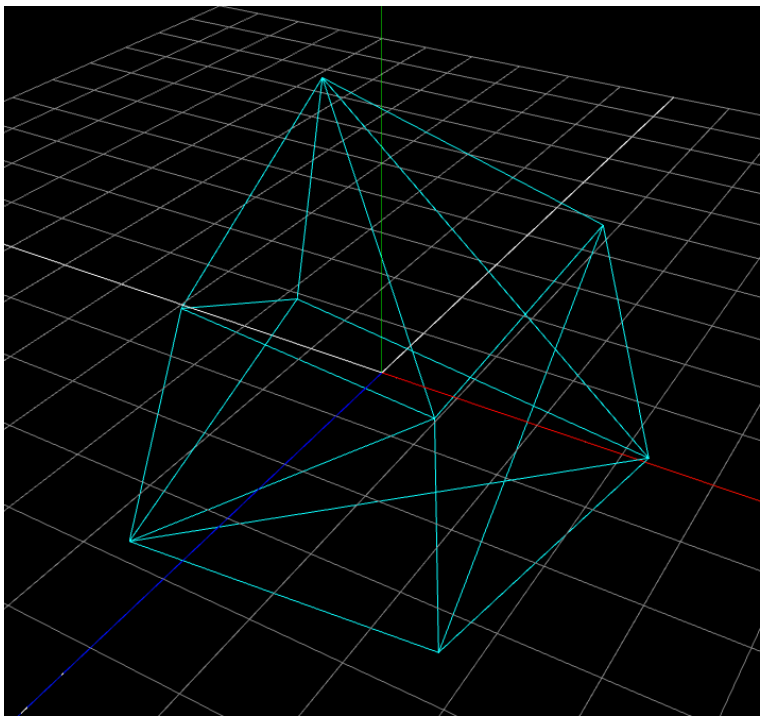

Figure 5

As you can see in Figure 5, the cube now rotates.

4 Different Render Modes

Switching render modes is done in the *handleKeyDown*() function, as seen in Figure 6. Each time either v, e, f are pressed, a new *CubeGeometry* is generated. Faces are generated with the same code as requirement 1. The edges are drawn using *EdgeGeometry* and vertices are drawn by using a *PointsMaterial* along with *CubeGeometry* in the mesh. Each keypress starts with *scene.remove*(*cube*), then the changes are made, then the cube is readded to the scene.

```
// Render modes.
case 70: // f = face
    scene.remove(cube);
    var boxGeo = new THREE.CubeGeometry(2,2,2);
    var boxMat = new THREE.MeshBasicMaterial({color: 0xfffff});
    cube = new THREE.Mesh(boxGeo,boxMat);
    scene.add(cube);
    break;

case 69: // e = edge
    scene.remove(cube);
    var boxGeo = new THREE.CubeGeometry(2,2,2);
    var edgeGeo = new THREE.EdgesGeometry(boxGeo);
    var edgeMat = new THREE.LineBasicMaterial( { color: 0xfffff } );
    cube = new THREE.LineSegments(edgeGeo, edgeMat);
    scene.add(cube);
    break;

case 86: // v = vertex
    scene.remove(cube);
    var boxGeo = new THREE.CubeGeometry(2,2,2);
    var vertMat = new THREE.PointsMaterial({color: 0xfffff, size: 0.1});
    cube = new THREE.Points(boxGeo,vertMat);
    scene.add(cube);
    break;
```

Figure 6

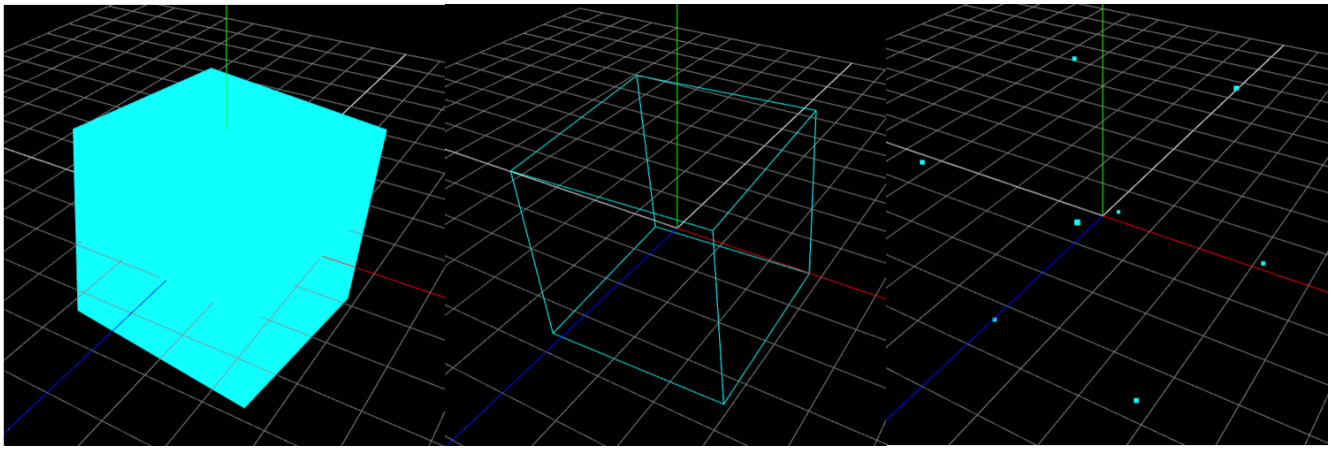

Figure 7

5 Translate the Camera

The camera is translated on its local axes using the arrow keys, as seen in Figure 8. Backwards/forwards zooming is handled by the scroll wheel, as seen in Figure 10. In addition to the function shown in Figure 10, A new *EventListener* was added for the mouse wheel, just under the *EventListener* for keypresses.

```
case 38: //Translate Camera UP (UP Arrow Key)
    camera.translateY(1);
    break;

case 40: //Translate Camera DOWN (DOWN Arrow Key)
    camera.translateY(-1);
    break;

case 37: //Translate Camera LEFT (LEFT Arrow Key)
    camera.translateX(-1);
    break;

case 39: //Translate Camera RIGHT (RIGHT Arrow Key)
    camera.translateX(1);
    break;
```

Figure 8

```
//Handle scrolling
function handleScroll(event){
    camera.translateZ(event.deltaY);
}
```

Figure 9

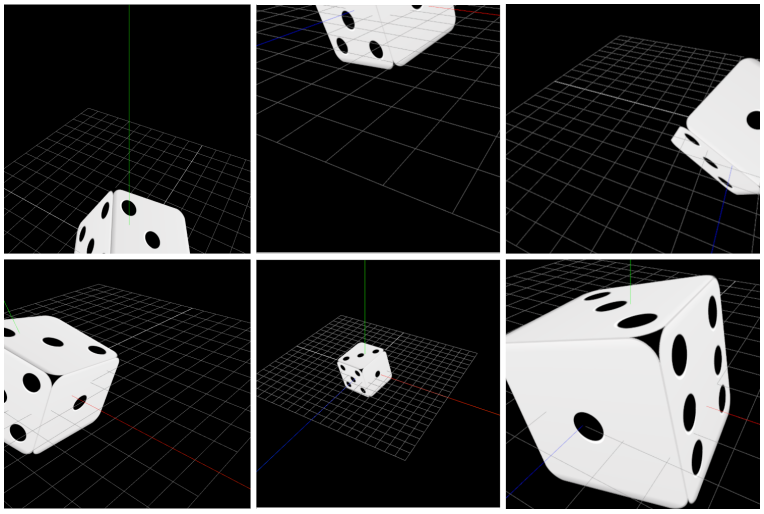

Figure 10

6 Orbit the camera

Camera orbiting is done using a click-and-drag motion. The button press is recorded when the mouse is pressed, and as the mouse moves, the spherical coordinates of the camera are recorded. The new position of the camera is then calculated using a triangle within the sphere; the hypothenuse being the radius of the circle. We have then have our two angles of rotation, theta and phi; theta is the angle between the plane $y = 0$ and the radius, phi is the angle between the plane $x = 0$ and the radius.

```
//Handle scrolling
function handleScroll(event){
    camera.translateZ(event.deltaY);
    lookAt.z += event.deltaY;
}

function handleMouseDown(event){
    mouseButtonDown = event.button;
    mousePosX = event.clientX;
    mousePosY = event.clientY;
    event.preventDefault();
}

function handleMouseMove(event){
    var cam = camera.position;
    var radius = cam.distanceTo( lookAt );

    //spherical coordinates
    var theta = Math.acos(cam.y/radius);
    var phi = Math.atan2(cam.z,cam.x);

    if(mouseButtonDown == 0){
        //we can increase/decrease theta and phi to move the camera about the sphere
        phi += (event.x - mousePosX)/100;
        theta += (event.y - mousePosY)/100;
        theta = Math.min(Math.PI - 10**(-6), Math.max(10**(-6), theta));

        cam = new THREE.Vector3(radius*Math.sin(theta)*Math.cos(phi), radius*Math.cos(theta), radius*Math.sin

        camera.position.copy( lookAt.clone().add(cam) );

        mousePosX = event.x;
        mousePosY = event.y;
        mousePosX = event.clientX;

        camera.lookAt(lookAt);
    }
}

function handleMouseUp(event){
    mouseButtonDown=-1;
}
```

Figure 11

As you can see in Figures 12,13 and 14, the camera is able to move freely.

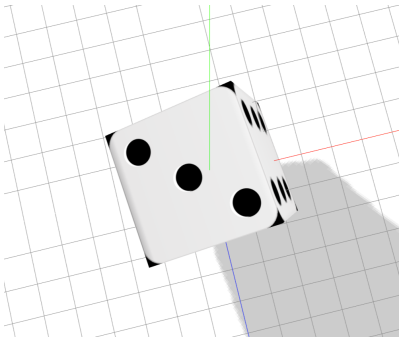

Figure 12

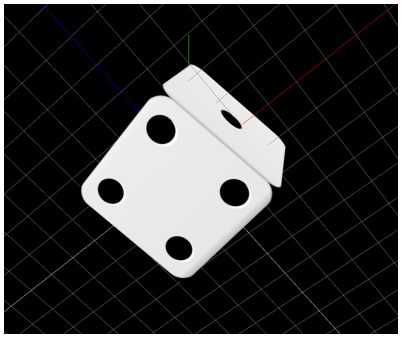

Figure 13

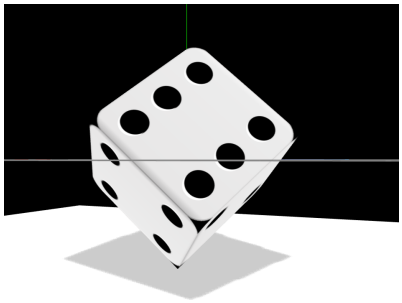

Figure 14

Unfortunately, I was not able to fully figure out how to adjust the lookAt position based on the translation of the camera, and the functionality is a little bit patchy and it tends to get into some form of gimbal lock. I suspect this is because when I translate the camera, I simply change the respective lookAt axis coordinate by the same amount, however the camera is moving on its local axes, whereas lookAt is on the global axes. If I had time I would be able to properly implement this.

7 Texture mapping

Textures are loaded using the built-in *TextureLoader*, then mapped onto each face of the cube. The function *createDiceTexture*() is then called from *init*(), as well as every time the user switches

to face rendering mode. The dice texture is seen in Figure 16.

```
function createDiceTexture()
{
    var texLoader = new THREE.TextureLoader();

    var face1 = texLoader.load('./textures/1.png');
    var face2 = texLoader.load('./textures/2.png');
    var face3 = texLoader.load('./textures/3.png');
    var face4 = texLoader.load('./textures/4.png');
    var face5 = texLoader.load('./textures/5.png');
    var face6 = texLoader.load('./textures/6.png');

    var faces = [
        new THREE.MeshBasicMaterial({map: face1}),
        new THREE.MeshBasicMaterial({map: face2}),
        new THREE.MeshBasicMaterial({map: face3}),
        new THREE.MeshBasicMaterial({map: face4}),
        new THREE.MeshBasicMaterial({map: face5}),
        new THREE.MeshBasicMaterial({map: face6})
    ]

    var boxMat = new THREE.MeshFaceMaterial(faces);
    return boxMat;
}
```

Figure 15

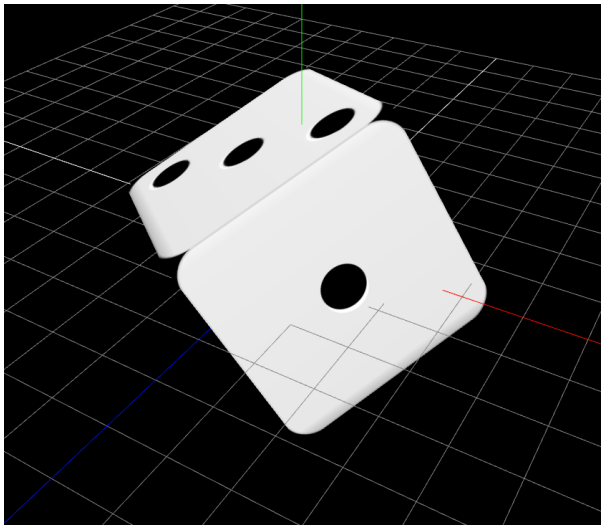

Figure 16

8 Load a mesh model from .obj

To load the Object from the .obj file, I chose to switch to a 'module' type script, as this was the easiest way to include *OBJLoader*. As a result, I had to download and include *three.module.js* instead of the original *three.js* file provided. Additionally, I took the most recent *OBJLoader.js* version from the project's GitHub. 2 import statements had to be included at the start of the script, as can be seen in Figure 17.

```
import * as THREE from './js/three.module.js';
import { OBJLoader } from './js/OBJLoader.js';
```

Figure 17

I then created the function *loadBunny*(), seen in Figure 18. It uses *OBJLoader*'s *load*() function, which takes the name of the .obj file as an argument, as well as function to be run upon loading. There are optional parameters to include functions for errors and such, but I chose not to include these. The object was then scaled to fit inside the cube by setting the *scale* property for all dimensions. I chose 0.3 as the scale factor as this seemed to work. The new object is then loaded into the scene as usual with *scene.add*(). I set $bunny = object$ as *bunny* is a global variable, which comes in handy later when performing operations on the object for the different rendering modes stated in Requirement 9.

```
function loadBunny(renderMode){
    loader.load(
        'bunny-5000.obj',
        function ( object ) {
            if(renderMode == 'e'){
                object.traverse( function ( child ) {

                    if(child.material)

                    {
                        child.material.wireframe = true;
                        child.material.color.setHex(0xffffff);
                    }

                } );
            }
            else if(renderMode == 'v'){
                var bunnyMaterial = new THREE.PointsMaterial( { color: 0xffffff , size: 0.01 } );
                object = new THREE.Points(object.children[0].geometry, bunnyMaterial);
            }
            object.scale.x = object.scale.y = object.scale.z = 0.3;
            scene.add( object );
            bunny = object;
        }
    );
}
```

Figure 18

9 Rotate the mesh, render it in different modes

The bunny is rotated in exactly the same way as the cube; a function *rotateBunny*() is created, the same as Figure 4, but with *bunny* instead of *cube*. This new function is then included in *animate*(). Rendering is done by the *loadBunny*() function, which is called with an optional parameter in the same location the rendering modes for the cube are done. For face mode, no parameter is included so the bunny is just loaded in. For edges, 'e' is passed, in which case after loading, the object is traversed and the materials of the children are set to wireframe (and colour set to blue too). For vertices mode, a *PointsMaterial* is created the same as for cube, and the object is set to a *Points* object, with the objects mesh geometry used.

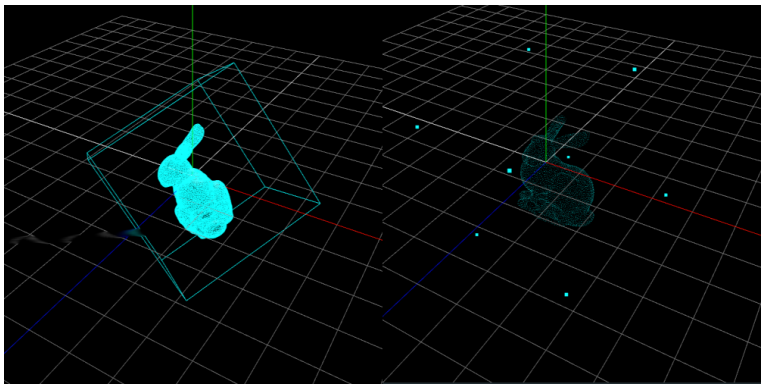

Figure 19

10 Be creative - Shadows

For Requirement 10 I decided to create a plane and cast shadows from the cube onto it. The plane was created, and then placed horizontally at $y = -2$, so that it wouldn't overlap the cube. A light source was then created, and a few settings were changed such as on the renderer and the cube so that they would render or cast/receive shadows. The result is Figure 22.

```
//adding plane
var planGeo = new THREE.PlaneGeometry( 20, 20, 20 );
var planMat = new THREE.MeshPhongMaterial( { color: 0xcccccc } );
var plane = new THREE.Mesh(planGeo,planMat);
plane.rotateX(-Math.PI * 0.5);
plane.position.set(0,-2,0);
plane.receiveShadow = true;
scene.add(plane);
```

Figure 20

```
//Light to cause shadows
var light = new THREE.SpotLight(0xffffff);
light.position.set(-3, 4, -5);
light.castShadow = true;
light.intensity = 1;
scene.add(light);
```

Figure 21

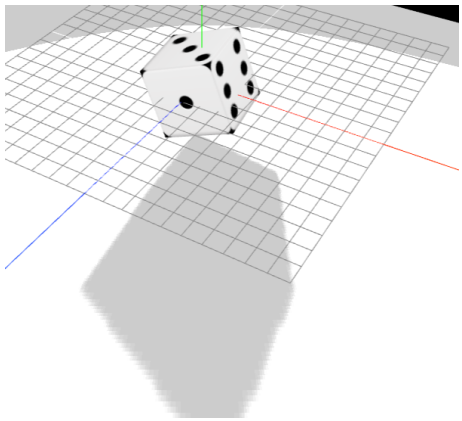

Figure 22

11 Evaluation

Given more time, I would be able to improve the general code quality as there is little code reuse currently, and implement more solid test coverage. I would also be able to fix the issues mentioned under Requirement 6.