CM20219: Fundamentals of Visual Computing

Coursework Part 2 – 2020/2021

INTRODUCTION

WebGL and three.js were used in conjunction with a web browser to complete this assignment. WebGL is a more dedicated implementation of the well-known platform OpenGL. JavaScript has an open-source API library known as three.js that allows for 3D computer graphics, with a multitude of features such as, but not limited to:

- Geometry and Modelling
- Transformations
- Viewing
- Lighting and Shading
- Texture Mapping
- Pixel Processing (Angel, 2017)

INTRODUCTION TO TASKS

This assignment was begun in a template provided by the lecturer. The template sets up a perspective camera facing the origin at position (3, 4, 5); creates a wire mesh in the x-z plane and creates an ambient source of lighting (this is not present in the final version of the project for reasons later specified). All other features in the project have been added in by the author.

NOTATION USED

Coordinates

All coordinates are dictated as follows: (x, y, z) where y, x, and z form the vertical, a horizontal and a perpendicular horizontal, respectively.

Code

Seldom code terminology will be included in descriptions with a different font and font size to make it clearer what is being referred to. For example: Here is the `code_snippet` in question.

REQUIREMENT 1: CREATING A SIMPLE CUBE

1.1 Specifications

- Cube must be centered at origin
- Must have opposite points at (-1, -1, -1) and (1, 1, 1)
- Must have faces orthogonal to x, y, z axes

1.2 Code

```
const newGeometry = new THREE.BoxGeometry(2, 2, 2);
const geometry = new THREE.BufferGeometry().fromGeometry(newGeometry);
const material = new THREE.MeshBasicMaterial();
cube = new THREE.Mesh(geometry, material);
scene.add(cube);
```

1.3 Result

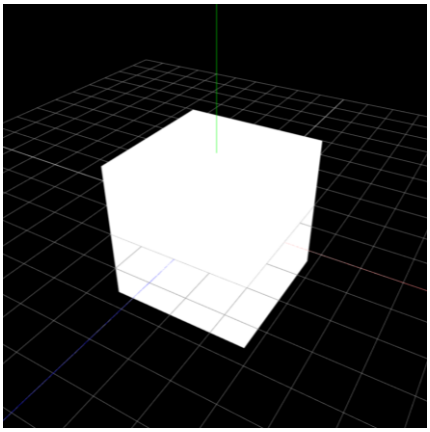

Figure 1: A simple cube generated in a web browser.

1.4 Description

A cube with dimensions 2 x 2 x 2 for height, width, length, respectively, is created at the origin. The code runs this block once when the page is loaded up in the `init()` function. Cube has a `MeshBasicMaterial` material loaded, which does not allow for shadows and is not affected by light sources (Mrdoob, 2020). Although it is difficult to discern, the faces are in fact orthogonal to the x-, y-, and z-axes, and so satisfy the specification of this task.

REQUIREMENT 2: DRAW COORDINATE SYSTEM AXES

2.1 Specifications

- Draw colour lines to represent axes
- Use RGB for XYZ values

2.2 Code

init:

```
displayAxis();
```

displayAxis:

```
// y-axis
const yaxismaterial = new THREE.LineBasicMaterial({ color: "rgb(0, 255, 0)" });
const ypoints = [];
ypoints.push(new THREE.Vector3(0, 0, 0));
ypoints.push(new THREE.Vector3(0, 5, 0));
const ylinegeometry = new THREE.BufferGeometry().setFromPoints(ypoints);
const yline = new THREE.Line(ylinegeometry, yaxismaterial);
scene.add(yline);
```

2.3 Result

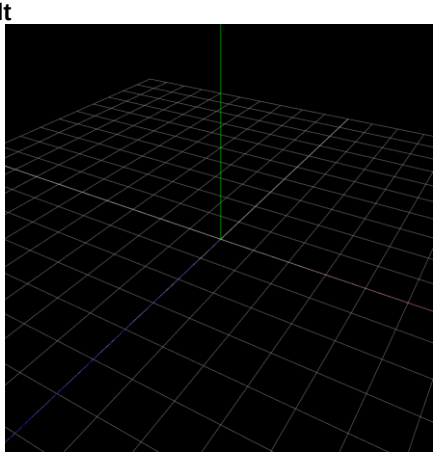

Figure 2: Coordinate axes generated in RGB values.

2.4 Description

The code snippet provided does not contain the full function, but rather a block that is repeated twice for each axis. A faint outline can be seen for each of the axis generated in Figure 2. Each axis is 5 units long to maintain visibility of each dimension without being overbearing to overall testing. First a new material is made, depending on the axis that is being considered: so red for X; green for Y; and blue for Z. Then a small array containing the origin point and a point 5 units along the axis, is created for each axis. Then a geometry connects the two with a line which is passed as a geometry along with the defined material to x-, y- or z-line variable to be added to the scene.

REQUIREMENT 3: ROTATE THE CUBE

3.1 Specification

- Rotate the cube about the x-, y- and z- axes respectively
- The axis generated from previous tasks and camera must not move

3.2 Code

Global variables:

```
var xRotation, yRotation, zRotation;
```

init:

```
xRotation = 0; yRotation = 0; zRotation = 0;
```

animate:

```
function animate() {
  requestAnimationFrame(animate);
  if (xRotation == 1){cube.rotation.x += 0.01;}
  if (yRotation == 1){cube.rotation.y += 0.01;}
  if (zRotation == 1){cube.rotation.z += 0.01;}
  renderer.render(scene, camera);}
```

handleKeyDown:

```
function handleKeyDown(event) {
switch (event.keyCode) {
    case 88: // x = rotate x axis
        xRotation = (xRotation == 1) ? 0 : 1;
        break;
    case 89: // y = rotate y axis
        yRotation = (yRotation == 1) ? 0 : 1;
        break;
    case 90: // z = rotate z axis
        zRotation = (zRotation == 1) ? 0 : 1;
        break;
```

3.3 Result

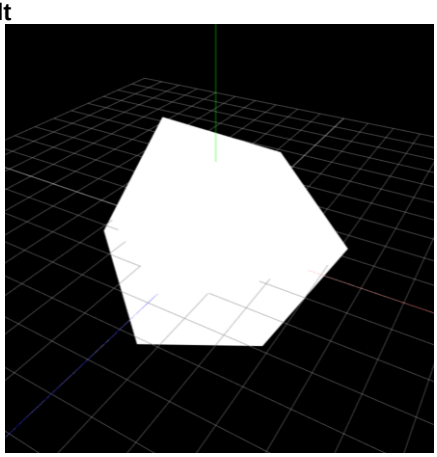

Figure 3: Cube is rotated on all axes.

3.4 Description

As is evident from Figure 3, the coordinate axes and camera stay fixed at the same angle while the cube has changed its position. The cube's center stays at the origin while the rest of the shape is transformed. To rotate the cube, code must be executed in the render function rather than initialization as it must be continuous and respond to input actively. Hence, a set of Boolean variables were created that check whether rotation is turned on for the cube in each axis: when the relevant key is pressed down, the "keydown" event triggers the Boolean to switch value with a shorthand if statement. Changing the value to 0 stops rotation in that axis where changing the value to 1 starts rotation in that axis.

REQUIREMENT 4: IMPLEMENT RENDER MODES

4.1 Specification

- Implement a keyboard shortcut
- Vertex mode must display all 8 vertices of the cube
- Edge render mode shows edges of primitive
- Face render mode shows 6 faces of the cube

4.2 Code

Custom lighting:

```
const light = new THREE.PointLight({ color: 0xff0000, decay: 2
 });
light.position.set(4, 5, 6);
light.castShadow = true;
```

```
scene.add(light);
```

Cubes:

```
const newGeometry = new THREE.BoxGeometry(2, 2, 2);
const geometry = new THREE.BufferGeometry().fromGeometry(newGe
ometry);
  const material = new THREE.MeshLambertMaterial();
cube = new THREE.Mesh(geometry, material);
  scene.add(cube);
edgeCube = new THREE.Mesh(geometry, new THREE.MeshBasicMateria
l({ color: "rgb(0, 255, 0)", wireframe: true }));
vertexCube = new THREE.Points(geometry, new THREE.PointsMateri
al({ color: "rgb(0, 0, 255)", size: 0.1 }));
```

Updated Animate function:

```
if (xRotation == 1){
      cube.rotation.x += 0.01;
      edgeCube.rotation.x += 0.01;
      vertexCube.rotation.x += 0.01;}
  if (yRotation == 1){
      cube.rotation.y += 0.01;
      edgeCube.rotation.y += 0.01;
      vertexCube.rotation.y += 0.01;}
  if (zRotation == 1){
      cube.rotation.z += 0.01;
      edgeCube.rotation.z += 0.01;
      vertexCube.rotation.z += 0.01;}
  renderer.render(scene, camera);
```

In handleKeyDown(event):

```
function handleKeyDown(event) {
  switch (event.keyCode) {
      // Render modes.
      case 70: // f = face
          scene.remove(edgeCube);
          scene.remove(vertexCube);
          scene.add(cube);
          break;
      case 69: // e = edge
          scene.remove(cube);
          scene.remove(vertexCube);
          scene.add(edgeCube);
          break;
       case 86: // v = vertex
          scene.remove(cube);
          scene.remove(edgeCube);
          scene.add(vertexCube);
          break;
```

4.3 Results

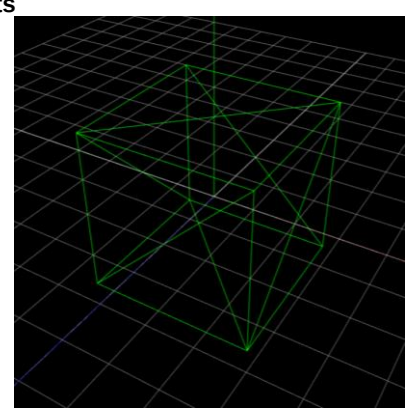

Figure 4: Edge render of cube showing wireframe of mesh.

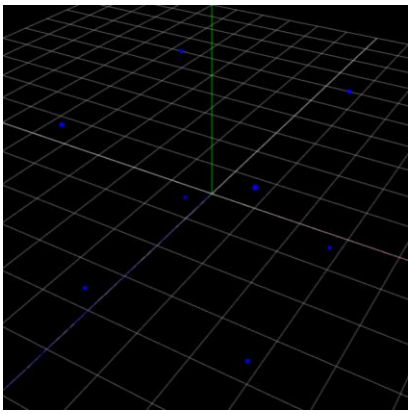

Figure 5: Vertex render mode showing 8 points of cube.

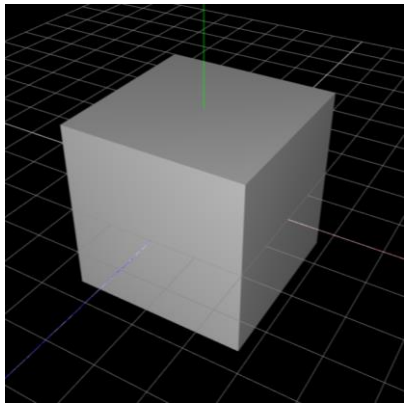

Figure 6: Updated face render mode with lighting.

4.4 Description

The provided ambient lighting, which provides a uniform level of lighting for all surfaces and from every direction had to be removed in order to make the cube clearer. In its stead, a new lighting system was implemented and set behind the camera to provide a lighting which feels natural. The light can cast shadows and as such has parameters enabled to do so.

`MeshBasicMaterial` is replaced with `MeshLambertMaterial` to allow for lighting to affect the surfaces of the cube. Two new cubes are made with the same geometry, but different

materials to create different render modes. As rotation must be constant across the different cubes, and allow for seamless transitioning between the three cubes, all three cubes are loaded initially. While one cube is visible, the other two have rotation values replicated if the visible cube is rotating. Although this requires more processing power, the payoff outweighs the drawbacks. By implementing these systems, requirement 4 has been fully answered.

REQUIREMENT 5: TRANSLATE THE CAMERA

5.1 Specification

- Manipulate the camera's location translating it:
  - Up/down
  - Left/right
  - Forward/backward
- This must NOT traverse across axes of the global coordinate system, but rather the camera's local vectors

5.2 Code

```
case 38: // up arrow = translate camera forwards
      camera.translateZ(-0.5);
      break;
case 40: // down arrow = translate camera backwards
      camera.translateZ(0.5);
      break;
case 37: // left arrow = translate left
      camera.translateX(-0.5);
      break;
case 39: // right arrow = translate right
      camera.translateX(0.5);
      break;
case 33: // PgUp = translate upwards
      camera.translateY(0.5);
      break;
case 34: // PgDn = translate downwards
      camera.translateY(-0.5);
      break;
```

5.3 Result

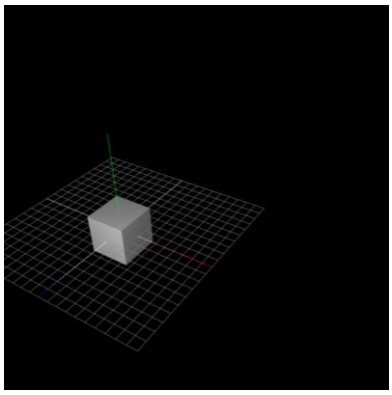

Figure 7: Camera translated further away from cube.

Description

The `handleKeyDown` subroutine is further developed with arrows and PgUp/PgDn detection for the three directions of travel. As the subroutine is called whenever the relevant key is pressed or being held down, the subroutine is either called once or continuously. `translateX` is a convenient function as it obtains the x position of the object and adds or subtracts the value passed as a parameter and updates the value for the object and vice versa for Y and Z directions. In order to obtain keycode values for javascript for the switch case statement, an open-source project was used (Gámez, 2020). Therefore, requirement 5 has been completed.

REQUIREMENT 6: ORBIT THE CAMERA

6.1 Specification

- Get camera to rotate around a "look at" point.
- Implement the OrbitControl library without using OrbitControl.js
- Allow the look at point to be changed to any location within the scene

6.2 Code

```
case 65: // a = rotate camera clockwise
      camera.translateX(-0.4);
      camera.lookAt(scene.position);
      break;
case 68: // d = rotate camera anti-clockwise
      camera.translateX(0.4);
      camera.lookAt(scene.position);
      break;
case 87: // w = rotate camera forwards
      camera.translateY(0.4);
      camera.lookAt(scene.position);
      break;
case 83: // s = rotate camera backwards
      camera.translateY(-0.4);
      camera.lookAt(scene.position);
      break;
```

6.3 Results

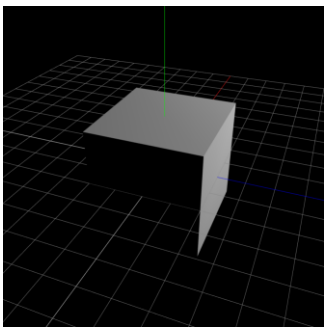

Figure 8: Horizontal camera rotation shows face of cube away from light.

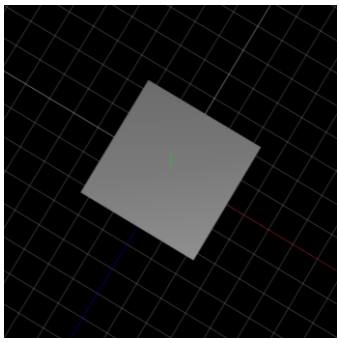

Figure 9: Vertical camera rotation shows top face of cube.

6.4 Description

This task was completed by further expanding on the keypress switch case statement by adding more cases with the help of the keycode reference (Gámez, 2020). By pressing the relevant key once, the camera is translated, and the camera rotation is changed with `lookAt` so that the camera turns to face the object. In turn a near-circular object is generated, performing a fixed distance orbit around the object, see Figure 10 for better visualization of this effect, where a red circle has been overlaid to represent the orbit.

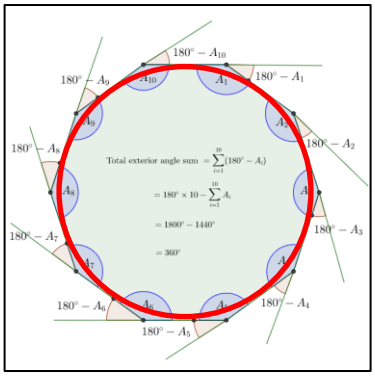

Figure 10: A decagon with exterior sides. (Jain, 2020)

The object being orbited around would be at the center of the circle. The unit 0.4 was chosen for demonstration purposes, but in practice a lower number would be better. The higher the number of `lookAt` functions per unit travelled, the more circular and accurate the orbit becomes. A way to implement this is to reduce the number in the brackets, as it decreases the unit distance travelled directly by the camera before it calls `lookAt` again. However, this would make camera rotation too slow and impractical, so a compromise was found. This approach also has a significant amount of synergy with the previous tasks to translate the camera, as it keeps the radius of rotation intact. Rotating vertically makes the camera bump into the y axis, and when it crosses over, the translation turns to the other side and translates the camera back. In short, crossing the y axis is impossible and an inverted rotation around the cube is also not possible with this method. To change the rotation point, `scene.position` can be translated within the parameter with another vector to set the focal point as something else within the scene. In this way, requirement 6 was answered.

On further inspection of the three.js library, in particular the implementation of `lookAt`, the object is rotated so that the internal z axis points towards the target vector. But of course, the target can spin around the z axis if that were the case, so a normal vector (up) is provided so that the internal y axis of the object lies on the plane of that up vector and internal z axis. One important feature of the camera in three.js is that it looks down the negative z axis rather than looking up towards the positive z axis. This is due to implementation and ease of use rather than anything else.

REQUIREMENT 7: TEXTURE MAPPING

7.1 Specification

- The cube has a texture applied to it
- Correct texturing (without skew) and perspective rendering
- Each face should look different

7.2 Code

```
const loader = new THREE.TextureLoader();
const materials = [new THREE.MeshLambertMaterial({ map: loader
.load('/img/img1.jpg') }), new THREE.MeshLambertMaterial({ map
: loader.load('/img/img2.png') }), new THREE.MeshLambertMateri
al({ map: loader.load('/img/img3.jpg') }), new THREE.MeshLambe
rtMaterial({ map: loader.load('/img/img4.jpg') }), new THREE.M
eshLambertMaterial({ map: loader.load('/img/img5.jpg') }), new
 THREE.MeshLambertMaterial({ map: loader.load('/img/img6.jpg')
})];
```

7.3 Results

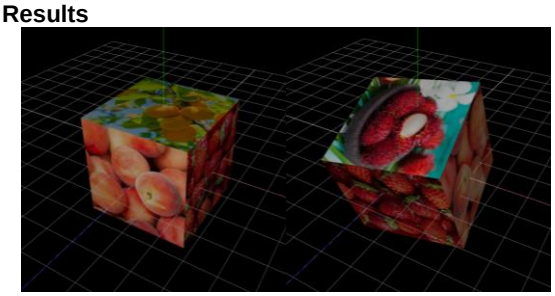

Figure 11: Fruit textured cube faces and point lighting

7.4 Description

In order to texture the cube, the previously configured lighting settings were used as these worked well with the blank textured cube. Instead of using a single material, an array containing 6 independent materials was created and used as a parameter for creating the cube. An in-built `TextureLoader` library was used to load images into the array. A discovery was made that the textures would often end up skewed or not proportionate, so all images were cropped beforehand into a square ratio so that they could properly load onto the cube.

REQUIREMENT 8: LOAD A MESH MODEL FROM .OBJ

8.1 Specification

- Load and display a mesh model (Stanford bunny)
- Scaled and translated to fit inside the cube

8.2 Code

```
var objLoader = new THREE.OBJLoader();
  objLoader.load('bunny-5000.obj', function (object) {
      object.scale.set(0.3, 0.3, 0.3);
      scene.add(object);
      bunny = object;
      object.traverse(function(child){
          if (child.isMesh){
child.material = new THREE.MeshLambertMaterial({color: 'rgb(25
5, 255, 0)'});}});});
```

8.3 Results

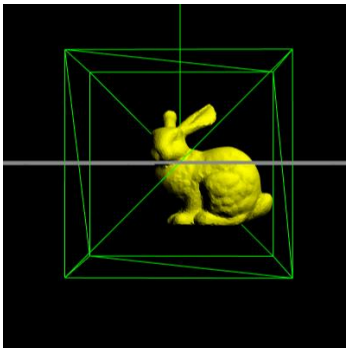

Figure 12: Stanford bunny face view.

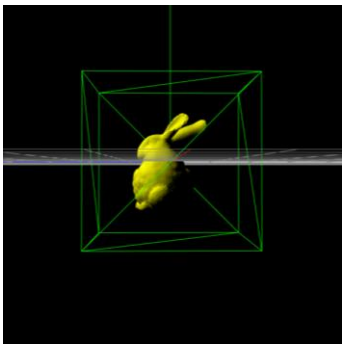

Figure 13: Stanford bunny side view.

8.4 Description

To load the Stanford bunny, a library provided with the assignment was used, called OBJLoader.js. `ObjectLoader.js` is an API that allows users to manipulate and load 3D objects from ".obj" files. As such, it had to be imported by including it as a source script in the HTML and defining the program script to be of type "module". Otherwise, importing would result in nothing occurring and exceptions being thrown, respectively. Once the object is loaded, the onload function is called. This in turn sets the scale to be a third of its original and adds the default bunny to the scene.

REQUIREMENT 9: ROTATE AND RENDER MESH

9.1 Specification

- Rotate model around x-, y- and z- axis
- Show the vertex rendering mode
- Show the edge rendering mode
- Show the face rendering mode (with materials, lighting and shading)

9.2 Code

Init:

```
wireBunny = new THREE.LineSegments(new THREE.WireframeGeometry
(child.geometry), new THREE.LineBasicMaterial({color: 'rgb(0,
255, 0)'}));
wireBunny.scale.set(0.3, 0.3, 0.3);
vertexBunny = new THREE.Points(bunny.children[0].geometry, new
 THREE.PointsMaterial({size: 0.01, color: 'rgb(0, 0, 255)'}));
vertexBunny.scale.set(0.3, 0.3, 0.3);
```

Render:

```
if (bunnyRot == 1) {
      bunny.rotation.y += 0.01;
      wireBunny.rotation.y += 0.01;
      vertexBunny.rotation.y += 0.01; }
```

handleKeyDown:

```
case 73: // i = rotate bunny
      bunnyRot = (bunnyRot == 1) ? 0 : 1;
      break;
case 85: // u = edge bunny
      scene.add(wireBunny);
      scene.remove(vertexBunny);
      scene.remove(bunny);
      break;
case 74: // j = face bunny
      scene.add(bunny);
      scene.remove(vertexBunny);
      scene.remove(wireBunny);
      break;
case 77: // m = vertex bunny
      scene.add(vertexBunny);
      scene.remove(bunny);
      scene.remove(wireBunny);
      break;
```

9.3 Results

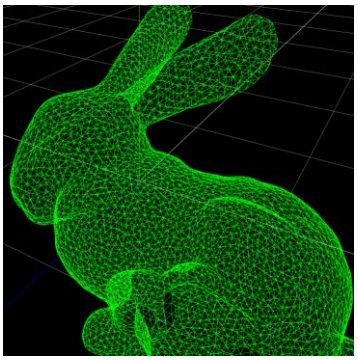

Figure 14: Wireframe render of imported bunny.

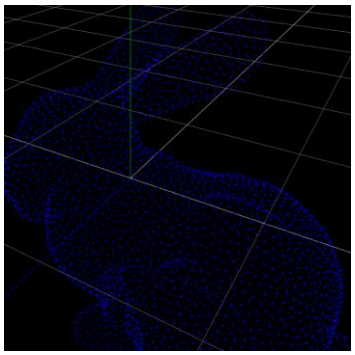

Figure 15: Vertex render of imported bunny.

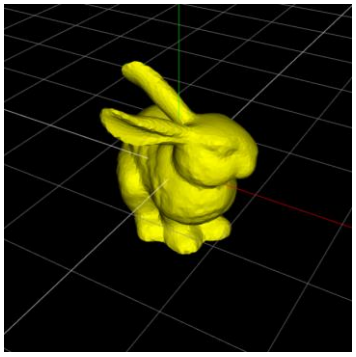

Figure 16: Stanford bunny is rotated along the y-axis.

9.4 Description

Two more objects had to be created with their own materials for edge rendering and vertex rendering for the bunny object. It was crucial that both objects also had their scale reduced to the same level and behaved much in the same way that the three cubes interact. As such, all three bunny objects are rotated when the relevant key is pressed while only one bunny is visible.

In order to access the objects while outside the onload function, global variables had to be declared, like that of the cube so that in the render function, they could be removed and added to the scene as required. Rotation was implemented like that of the cube, where all three objects were rotated so that the position is not reset when render mode is changed.

REQUIREMENT 10:

10.1 Specification

- Create a ground plane which can receive shadows
- Should be able to control rotation of light source and elevation of ground plane through a GUI
- Should be able to toggle the plane and GUI on and off

10.2 Code

addPlane function:

```
  if (extension % 2 == 0) {
        gui.destroy();
        scene.remove(plane);
        scene.remove(directlight);
        scene.add(light);
  } else {
        scene.remove(light);
        plane = new THREE.Mesh(new THREE.PlaneGeometry(15, 15),
new THREE.MeshLambertMaterial({ color: 'purple', side: THREE.D
oubleSide }));
        plane.rotation.x += Math.PI / 2;
        plane.position.y -= 2;
        plane.receiveShadow = true;
        scene.add(plane);
        directlight = new THREE.DirectionalLight('white', 1.5);
        directlight.position.set(3, 4, 5);
        directlight.castShadow = true;
        scene.add(directlight);
        controls = new function(){this.angle = 45;
this.planeHeight = -2;}
        gui = new dat.GUI();
        gui.add(controls, 'angle', 0, 360);}}
        gui.add(controls, 'planeHeight', -10, -2);
```

Render:

```
if (extension % 2 == 1){
        directlight.position.x = Math.sqrt(34) * Math.sin((contr
ols.angle * Math.PI)/180);
        directlight.position.z = Math.sqrt(34) * Math.cos((contr
ols.angle * Math.PI)/180);
plane.position.y = controls.planeHeight;}
```

handleKeyDown:

```
case 81: // q = extension task
    extension++;
    addPlane();
```

10.3 Results

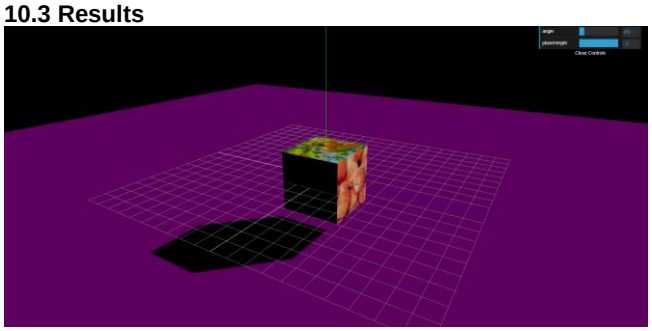

Figure 17: Cube casting shadow onto plane.

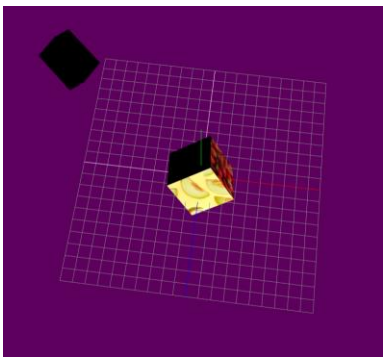

Figure 18: Rotated cube casting shadow on lower height plane.

10.4 Description

To begin, a plane had to be created so that shadows from the cube could be cast. It was created with a Lambert material so that shadows could be cast upon it. In order to create the lighting, the old lighting had to be removed, otherwise there would be two shadows from the same cube because of the two sources. In order to control the two elements, a GUI had to be implemented: `dat.GUI`. `dat.GUI` provides a framework that allows programmers to easily test values and simulations with data in their programs.

In order to add elevation to the plane, a `controls` variable was created to store all independent variables that needed to be changed for this requirement. In it, the variable `planeHeight` subtracts units from a base negative so that a meaningful range of heights can be tested. The property of `DirectionalLight` to be unaffected by rotation meant that the same approach for camera rotation could not be taken. By implementing the parametric equations of a circle, and converting the angle given by the GUI into Radians a circular orbit for the light source was created. Finally, to toggle the whole system on and off, instead of creating a variable switching between 0 and 1, an integer was simply incremented whenever the relevant key was pressed, and its value was checked for oddness. If the integer was odd, then the whole system was created and if not, the `DirectionalLight`, plane and GUI were all destroyed and original light added back into the scene.

LESSONS LEARNED, LIMITATIONS AND FUTURE WORK

Due to time constraints and resources available because of the pandemic, several concessions had to be made in order to finish the assignment. Initially the goal was to implement a form of physics engine within the renderer, letting the cube collide with the plane using gravity. Adding directional movement for the cube on the plane would have also been considered. Three.js and WebGL renderer have been very eye opening and helped demystify JavaScript. I have developed a solid foundation for understanding JavaScript within the browser environment and learnt debugging. Exposure to different libraries and opening source code for Three.js have showed me how detailed and meticulously planned code must be written for it to accepted into general use and collaboration. I aim to add Three.js to my list of skills and further explore it outside of curricular work.

REFERENCES

[1] Angel, Ed, 2017. The Case for Teaching Computer Graphics with WebGL: A 25-Year Perspective. IEEE Computer Graphics & Applications, 37(2), pp.106–113.

[2] Mrdoob. (2020). Three.js. OBJLoader. Available: https://threejs.org/docs/. [Accessed: 18/12/2020]

[3] Gámez, D. (2020). Key.js. Available: https://keyjs.dev/. [Accessed: 17/12/2020]

[4] Jain, Y. (2020). General Polygons- Angles. Available: https://brilliant.org/wiki/general-polygons-angles/. [Accessed 16/12/2020]