CM20219 – Coursework Part 2 – Report

For each of the ten requirements, I shall describe the mathematics involved and the implementation I used to achieve the requirement. I will include screenshots of code and testing when appropriate for each requirement. After that I will discuss the limitations of my implementation and how I would improve should I have to implement a similar program again in the future.

Requirement 1: Draw a Simple Cube:

The first requirement involved constructing a cube of dimensions 1x1x1 units centred on the origin and adding it to the scene. This would require a geometry object, and material object that would together form the mesh object that would represent the cube in the scene on the screen. All these objects are included in the three.js package. I used the following the code to implement the cube:

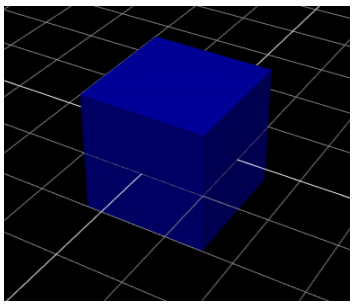
```
var cube, shape, material, cubeVertices, cubeEdges, cubeTextureMaterials;
```

First, I instantiated variables for the mesh object (cube), geometry (shape) and the material (material) that would form the mesh of the cube. These are instantiated in the global scope so that they may be accessed anywhere within the script.

```
// TO DO: Draw a cube (requirement 1).
shape = new THREE.BoxGeometry(1, 1, 1);
material = new THREE.MeshPhongMaterial({ color: 0x000077 });
cube = new THREE.Mesh(shape, material);
scene.add(cube);
```

After that, in the 'init' function, I defined these variables with the required values. The shape becomes a 'BoxGeometry' with parameters representing the dimensions of the cube (1x1x1); the material is a Phong material as opposed to a basic mesh material so we can make use of lighting later. Finally, cube object makes a 'mesh' of the cube using its geometry and material and is added to the scene.

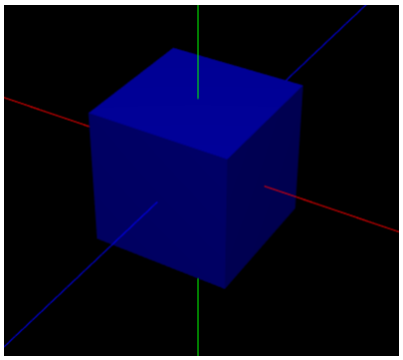
This produces the image shown right. A cube with dimensions 1x1x1 units of the material specified, as of now, a light blue cube and is centred on the origin (0,0,0) by default.



Requirement 2: Draw Coordinate System Axes:

The next requirement specified adding three lines representing the coordinate axes (x = 0, y = 0, z = 0) in the colours: red, green and blue, respectively. To implement this, I created a 'line' object for each of the three axes, with each having its own geometry (consisting of three points the line runs through) and material (describing the colour of the axis). Here is the code in question:

```
//create materials for each of the axis lines
var xMaterial = new THREE.LineBasicMaterial({ color: 0xff0000 });
var yMaterial = new THREE.LineBasicMaterial({ color: 0x00ff00 });
var zMaterial = new THREE.LineBasicMaterial({ color: 0x0000ff });

//create geometries for each axis object, three points that constitue their direction
var xGeometry = new THREE.Geometry(); xGeometry.vertices.push(new THREE.Vector3(-100, 0, 0));
var yGeometry = new THREE.Geometry(); yGeometry.vertices.push(new THREE.Vector3(0, -100, 0));
var zGeometry = new THREE.Geometry(); zGeometry.vertices.push(new THREE.Vector3(0, 0, -100));

//create each axis object, using their respective materials and geometries
var xAxis = new THREE.Line(xGeometry, xMaterial);
var yAxis = new THREE.Line(yGeometry, yMaterial);
var zAxis = new THREE.Line(zGeometry, zMaterial);
```



First, I defined the materials, geometries and then the lines. As you can see, each material describes a different colour, first red then green and blue. Also, for each line, I had to specify three points: represented by these THREE.Vector3 objects, taking units for each dimension. For each axis, I gave them the equivalent of 100 units backward and forward along their axis and the origin (0,0,0). Then I added them to the screen, giving the result shown to the right of the code. I have removed the grid-helper to make the lines easier to see.

Requirement 3: Rotate the Cube:

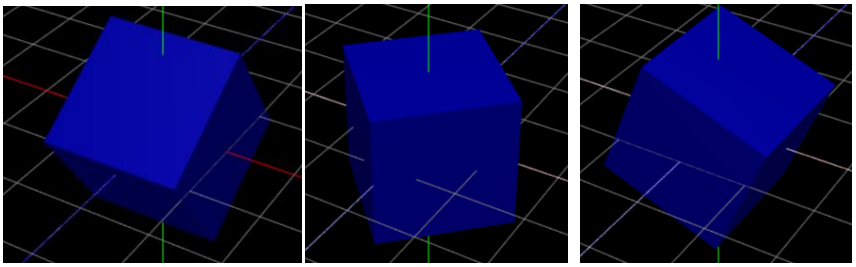
After the cube and axes have been added to the scene, we must now be able to fully control rotating the cube in each of the x, y and z axes. To implement this, I first created Boolean values for rotation on each axis, so that when the animate request that redraws the scene each frame is called, it would know to first rotate the cube in the corresponding direction(s).

```
var rotateX = false; var rotateY = false; var rotateZ = false;
```

The user can control the value of these Booleans, thus the rotation of the cube with button presses, which are defined in the keypress handling function. I assigned the '1', '2' and '3' keys to control the Boolean for the x, y and z axes respectively. The implementation required simply adding a switch case to determine which key is pressed and then change Boolean values accordingly, as shown on the left image below:

```
//rotations: '1' for x axis,
case 49: //'1'
    rotateX = !rotateX;
    break;
case 50: //'2'
    rotateY = !rotateY;
    break;
case 51: //'3'
    rotateZ = !rotateZ;
    break;
```

```
if (rotateX) {
    //rotate on x axis
    cube.rotation.x += 0.01;
```



Next, to handle when a Boolean value is set to true, I use 'if' statements for each Boolean value to check whether they are true and then rotate the cube accordingly. The images of cubes are from left to right: the cube rotated slightly in the x axis, then the y axis and finally, in the z axis.

Requirement 4: Different Render Modes:

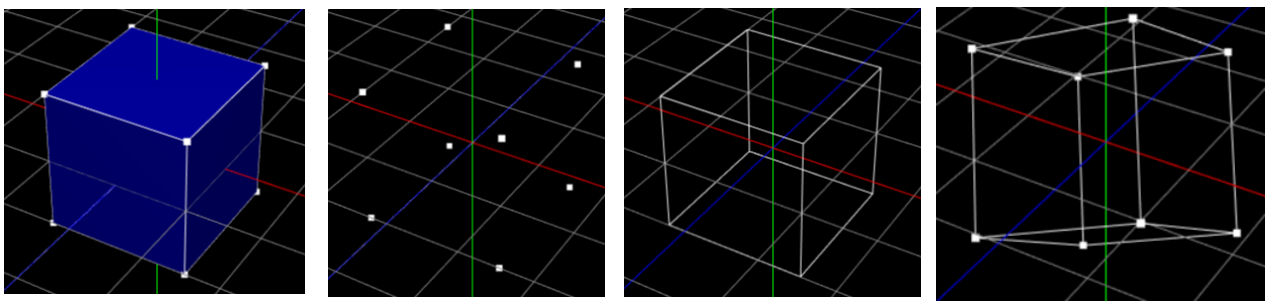
Fourth on the list of requirements, is adding the ability to render the cube in terms of just its vertices, edges and faces (how it currently appears with the help of lighting). To do this, I first instantiated variables for each of the corresponding objects: one to represent vertices, edges and faces (or the mesh as it is):

```
var cube, shape, material, cubeVertices, cubeEdges, cubeTextureMaterials;
```

These are the 'cube', 'cubeVertices' and 'cubeEdges' variables as shown previously, once more declared in a global scope so that they may be accessed anywhere within the script. Next, I would have to define each of these variables in the 'init' function, of which cube has already been defined.

```
cubeVertices = new THREE.Points(shape, new THREE.PointsMaterial({ color: 0xffffff, size: 0.1 }));
cubeEdges = new THREE.LineSegments(new THREE.EdgesGeometry(shape), new THREE.LineBasicMaterial({ color: 0xffffff
```

As shown above, the vertices object uses an existing three.js object known as 'Points' that takes the geometry of the object we want the vertices of (in this case, the predefined 'shape') and a material to go alongside it. For the material, I specified the vertices been shown as white and as squares of dimensions 0.1x0.1 units. The process was much the same for the edges object. I instantiated an existing three.js object for edges (or 'lines') which required the cube's geometry and a specified material, which I set as purely, the colour white. This gave the results below.



The images shown above from left to right show: the vertices and edges matching that of the cubes; the vertices exisiting as an object on its own rendering; the same as before but with edges and finally, the ability to rotate the edges and vertices in the same fashion as you can with the cube and its faces.

Requirement 5: Translate the Camera:

Now that we can render the cube and its vertices and edges, as well as being able to rotate each of those, we should now implement a way to translate the camera in a 3D space. This means being able to move the camera in each of the x, y and z axis forward and backward, positive and negative.

In terms of implementation, this was relatively easy and simple to do. I simply created new key press events for six keys, which would each translate the camera in one direction in one of the axes respectively. For the x and y axis, I chose the arrow keys as the felt a natural fit, as they are commonly associated with movement left and right, up and down, which is what translating the keys in the x and y directions would do. For the z axis, I chose the 'z' key and 'x' keys: 'z' for its namesake and 'x' for its convenient placement next to the 'z' key. Shown below is the code representing the above:

```
//CAMERA MOVEMENTS
case 38: //upArrow, move camera forward along Y axis
    camera.translateY(0.05); break;
case 40: //downArrow, move camera backward along Y axis
    camera.translateY(-0.05); break;
case 37: //leftArrow, Move camera backward along X axis
    camera.translateX(-0.05); break;
case 39: //rightArrow, move camera forward along X axis
    camera.translateX(0.05); break;
case 90: //z_key, move camera backward along Z axis
    camera.translateZ(-0.05); break;
case 88: //x_key, move camera forward along Z axis
    camera.translateZ(0.05); break;
```

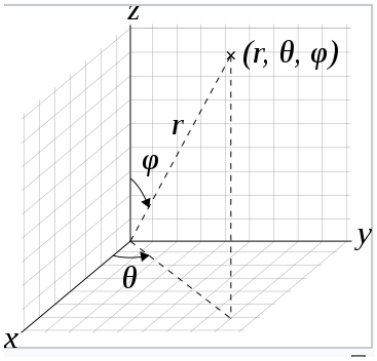
These were added to the event handler for key presses switch-case statement, wherein each case is a different key value. Each key translates the camera in a direction along an axis by 0.05 units.

Unfortunately, it is very difficult to show testing for camera translations in static screenshots, so I cannot include any evidence of it occurring, so you will have to trust my demonstration results.

Requirement 6: Orbit the Camera:

As well as being able to move the camera in the x, y and z axes, the user should be able to move the camera around the cube in an arc-ball mode, as if it were following the surface of a sphere around the cube.

To implement this, I would have to make use of mathematical equations to calculate spherical coordinates of the cube and move the camera alongside the sphere's surface.



[1]

[2]

$$x = r \sin\theta \cos\varphi$$
$$y = r \sin\theta \sin\varphi$$
$$z = r \cos\theta$$

```
//create values for start latitude and longitude
var radius = 0; var PHI = 0.0; var THETA = 0.0;
```

As the diagram above shows, you can calculate what would be the spherical coordinates in a Cartesian coordinate system (which the three.js camera uses). Wherein the latitude is represented by the angle theta (Θ) and the longitude represented by angle phi (φ). As well as this, the radius of the sphere is needed. I represented all these values as floating-point variables within my code, also defined in a global scope so that they may be accessed from anywhere within the script (code shown above).

[1] https://en.wikipedia.org/wiki/Spherical_coordinate_system#/media/File:3D_Spherical.svg
[2]https://wikimedia.org/api/rest_v1/media/math/render/svg/7d6175912d0e1731adf6b7ac6e14cc814355b659

I then created a function that would calculate the spherical coordinates needed to move the camera to in order to move the camera to that position in a Cartesian plane. I calculated the x, y and z values using the given equations above, x = r*sinΘ*cosφ and so on. The radius I would use would be the current distance the camera is directly from the origin (0,0,0), thankfully the camera object in three.js stores this value itself, so there was no need for me to calculate it myself. Here is the function I implemented, shown left, below:

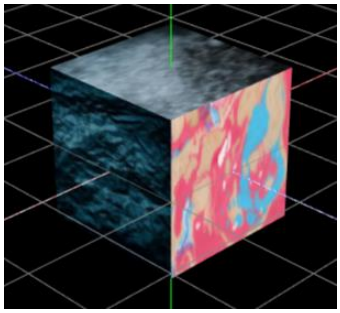
```
//function to calculate spherical coordinates to 'orbit' an object
function calculateSphericalCoordinates() {
    radius = camera.position.distanceTo(origin);

    camera.position.x = radius * -(Math.cos(PHI) * Math.cos(THETA));
    camera.position.y = radius * Math.sin(PHI);
    camera.position.z = radius * Math.cos(PHI) * Math.sin(THETA);
    camera.lookAt(origin);
}
```

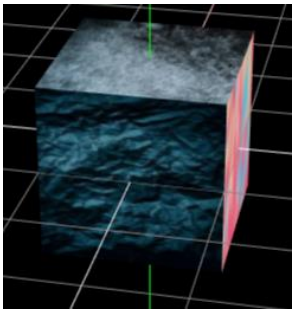
The function calculates and assigns each of the new values of x, y and z independently and then sets the camera to look at the origin to give the effect of 'orbiting' the centre.

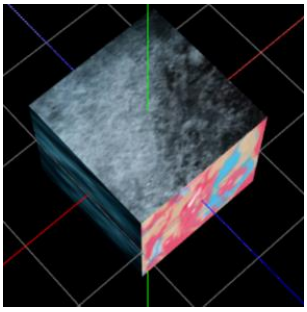
```
//ORBIT CONTROLS WASD
case 87: //w = orbit up longitude
    PHI += (3 / 180) * Math.PI;
    calculateSphericalCoordinates();
    break;
case 83: //s = orbit down longitude
    PHI -= (3 / 180) * Math.PI;
    calculateSphericalCoordinates();
    break;
case 68: //d = orbit right latitude
    THETA += (3 / 180) * Math.PI;
    calculateSphericalCoordinates();
    break;
case 65: //a = orbit left latitude
    THETA -= (3 / 180) * Math.PI;
    calculateSphericalCoordinates();
    break;
```

Now that we can calculate the new positions, we must add the ability for the user to control which directions the camera moves in when orbiting the cube, which is centred on the origin. This was implemented using the code shown right. It was the case of adding more key-press event switch-cases. I decided to use the 'w','a','s' and 'd' keys for this purpose as, like the arrow keys, they are also commonly used for user movement in applications. In this case, 'w' and 's' would move the camera up and down the sphere's surface, or the longitude and 'a' and 'd' would move the camera left and right along the sphere's surface, or the latitude.

I therefore added cases to the switch-case statement for each of the keys. To move in the desired direction, I would add or subtract to the value of phi (for longitude) or theta (for longitude) when that button is pressed, which would then call the calculateSphericalCoordinates() function, which calculate the new camera position and also moves the camera there.







The three images above show the use of the orbiting camera. I have used textures on the side of the cube to make it easier to tell the camera has moved to view different faces. The leftmost image is the cube at neutral position, i.e. where the camera is initially positioned. The image in the centre after moving the camera leftward, i.e along the latitude of the sphere created. This is why we can see less of the pinkish textured side and more of the dark blue texture. The rightmost image shows the view of the cube after moving the camera upward, i.e. along the longitude of the sphere created. This is why we can see less of the faces on the 'sides' of the cube and more of the grey texture on the 'top' face of the cube.

Requirement 7: Texture Mapping:

The next requirement would have us load and assign textures to different faces on the cube. To do this, I would have to make use of an inbuilt function of three.js, an object called a texture loader. Below is the code

used to assign this object to a variable, again declared in a global scope so that it may be accessed anywhere within the script.

```
var textureLoader = new THREE.TextureLoader();
```

Next, I would use a function of the texture-loader object to load the textures (stored as images) into variables, for which I used an array of six elements, one for each texture of each face. The array was called 'cubeTextureMaterials' and has been shown in code above, in a global scope with other properties of the cube. The code below illustrates calling the texture-loader object's function to load textures as images and assign them to materials which can be used for objects. Here I used basic mesh materials:

```
cubeTextureMaterials = [
    new THREE.MeshBasicMaterial({ map: textureLoader.load("1.jpg") }),
    new THREE.MeshBasicMaterial({ map: textureLoader.load("2.jpg") }),
    new THREE.MeshBasicMaterial({ map: textureLoader.load("3.jpg") }),
    new THREE.MeshBasicMaterial({ map: textureLoader.load("4.jpg") }),
    new THREE.MeshBasicMaterial({ map: textureLoader.load("5.jpg") }),
    new THREE.MeshBasicMaterial({ map: textureLoader.load("6.jpg") }),
];
```

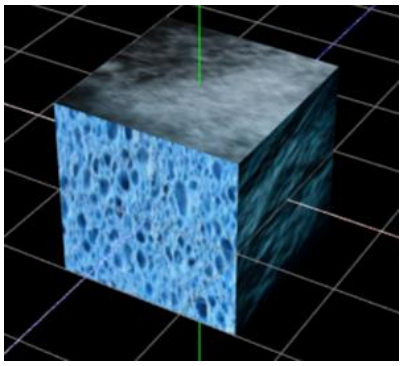
This array would then store the six textures. Now, I needed to assign a key-press to change the cube faces from pure block colour to these textures.

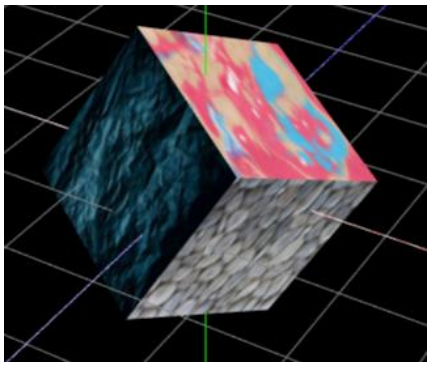
Similar to rotation, I created a Boolean value to store whether the cube is using textured faces or not, then when the key is pressed, switch the value of the Boolean and change the material used for the cube (textured or not). Shown left is the code for key-pressing and to the right, the Boolean value in a global scope:

```
//load textures for requirement 5
case 84: //t = texture
    if (textured) {
        cube.material = material;
    }
    else {
        cube.material = cubeTextureMaterials;
    }
    textured = !textured;
    break;
```

```
var textured = false;
```

To change the material used, you can call the three.js 'mesh' object representing the cube and change the value of its 'material' property. You can just assign the array storing the textures as images, and three.js will automatically assign one per face of the cube. Below are images showing some of the textures of the faces:





Here the cube has been rotated along various axes to show the different faces and textures.

Requirement 8: Load a Mesh Model from an Object File:

Aside from simple 3D shapes, you can also use three.js' inbuilt object-loader to load an object model from an object file (.obj extension). To implement this, I firstly declared various variables for the model similar to the cube; one for the mesh, material, geometry, vertices and edges respectively.

```
var bunny, bunMaterial, bunGeometry, bunVertices, bunEdges;
```

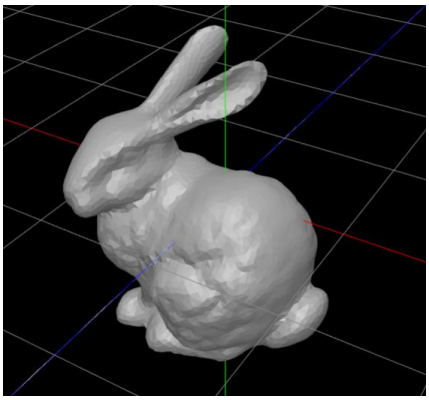
I have named the variables with variations of the word ‘bunny’ in them as the model we are loading is of a bunny-rabbit.

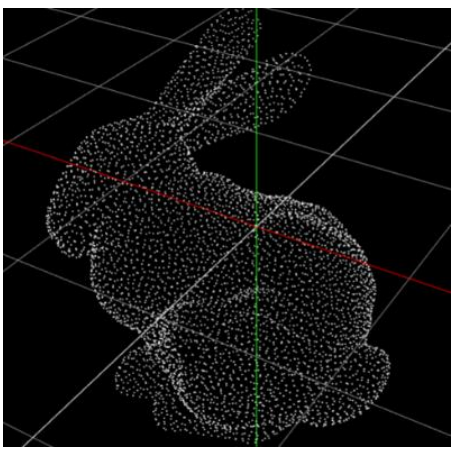
```
objLoader.load('bunny-5000.obj', function (object) {
    //form geometry, edges, vertices and faces from loaded object
    bunGeometry = object.children["0"].geometry;
    bunny = new THREE.Mesh(bunGeometry, new THREE.MeshPhongMaterial({color:'blue'}));
    bunVertices = new THREE.Points(bunGeometry, new THREE.PointsMaterial({ color: 0xffffff, size: 0.01 }));
    bunEdges = new THREE.LineSegments(new THREE.EdgesGeometry(bunGeometry), new THREE.LineBasicMaterial({ color: 0xffffff }));

    //create a box to scale object from and calculate the width, depth and height of the object
    var bunScaleBox = new THREE.Box3().setFromObject(bunny);
    var w = bunScaleBox.max.x - bunScaleBox.min.x;
    var d = bunScaleBox.max.y - bunScaleBox.min.y;
    var h = bunScaleBox.max.z - bunScaleBox.min.z;

    //scale the various objects, edges, vertices and faces to fit inside cube (1x1x1)
    bunny.scale.set(1 / w, 1 / d, 1 / h); bunVertices.scale.set(1 / w, 1 / d, 1 / h); bunEdges.scale.set(1 / w, 1 / d, 1 / h);

    bunny.position.x = -0.25;
    bunEdges.position.x = -0.25;
    bunVertices.position.x = -0.25;


})
```

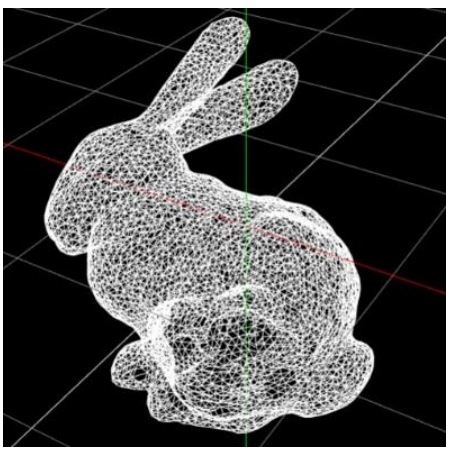
The next stage in implementing the model loading is to use the three.js object-loader object’s inbuilt method of loading external models which is relatively complex (shown in the code above). First we pass the filepath of the object file, called ‘bunny-5000.obj’. Then, we must define a function which is called after the pbject is sucessfully loaded. This function assigns the geometry of the loaded model into our associated variable ‘bunGeometry’. Then from this geometry, we assign the values of vertices, edges and meshes to the right variables. After this, I needed to scale the model to fit inside the 1x1x1 cube. First I created a box object included in the three.js package as this can surround the given model to represent it, then I calculated the width, height and depth of the model from the box object’s values. Finally, I used the in-built method of the three.js ‘object’ object to scale the rabbit model to fit in a 1x1x1 cube then translated the model slightly to centre it. Below to the right is the loaded model:

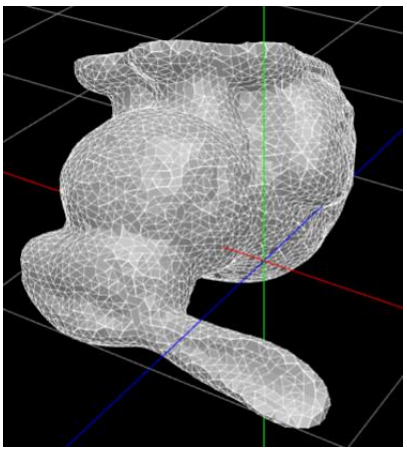
Requirement 9: Rotate and Render the Model:

As with the cube, we must be able to rotate the model in the x, y and z axes and render it in different modes. The implementation was near identical. It used the same implementation but swapped the variable names from the cube’s to the model’s, therefore I do not feel it necessary to include the code within this text, but below are examples of all these capabilities in action: From left to right: the model rendered only with vertices, then only with its edges, and finally, the model after rotation in various degrees in all axes.









Requirement 10: Something Creative:

The final requirement was the descriptive instruction: 'do something cool!'. Therefore, I decided to try and replicate the previously popular and controversial game 'flappy bird'. It is a simplistic game, whereby you control a bird 'flapping' up and down, dodging the incoming obstacles. If you collide with an obstacle, it's game over.

To implement this game, I created a new html file, called 'flappybird.html'. First, I declared all necessary values that I would need within the game:

```
var bird, birdGeometry, birdMaterial;
var pipe1, pipe2, pipe3, pipeGeometry, pipeMaterial;

var gravity = 0.001;
var velocity = 0.0;

var pipespeed = 0.02; var jumpModifier = -0.04;

var birdBox; var pipe1Box; var pipe2Box;

var score = 0;
```

This included: an object for the bird mesh, geometry and material; values to represent gravity and physics involved with the bird 'jumping'; values for the current speed at which the game moves; boxes to detect collision between objects and finally, a way to score the player.

Then, similar to all other objects in the previous scene, I had to declare and assign values to the materials and geometries of those objects and add them to the new scene used for this game.

The next part I implemented was the 'animate' function. Here is why I implemented the 'gravity' aspect of the bird 'flapping' up and down the screen.

```
if (bird.position.y > 0) {
    velocity += gravity;
    bird.position.y -= velocity;
}
else {
    bird.y.position = 0;
```

Here is the code showing 'falling'. Each frame, the value of downward velocity is increased by gravity, creating a downward acceleration and moving the bird ever-downward each frame. It only does this however if the bird is not touching the floor, i.e. its y position is greater than 0.

```
// Handle keyboard presses.
function handleKeyDown(event) {
    switch (event.keyCode) {
        case 32: //space for 'jump'
            velocity = jumpModifier;

            break;
    }
}
```

The player 'flaps' the bird upward by pressing the 'space' key. This is implemented using key presses like the previous script. When the space bar is pressed, the velocity is reset to the current 'jumpModifier'. This changes the movement of the bird from downward to upward and so the bird 'jumps' as gravity inevitably creates a negative velocity value again.

```
function detectCollision() {
    birdBox = new THREE.Box3().setFromObject(bird);
    pipe1Box = new THREE.Box3().setFromObject(pipe1);
    pipe2Box = new THREE.Box3().setFromObject(pipe2);

    if (birdBox.intersectsBox(pipe1Box) || birdBox.intersectsBox(pipe2Box) || bird.position.y <= 0) {
        alert("GAME OVER!!!\nScore: "+score);
    }
}
```

The next stage of the implementation was to tell if the bird was colliding with one of the obstacles, or 'pipes'. The three.js object Box3 has an in-built method to tell if two objects are colliding or not, so I made use of those. I first create a box for each object involved, and then use the in-built method to find out if they are colliding. If they are, or if the bird is touching the floor, the game is ended with an alert and the player is told their score. It is difficult to show testing for this game, as it involves acceleration and player key-presses, which cannot be seen with screenshots, so one must either trust my demonstration results or play the game yourself with the given code and instructions.

Discussion

Finally, I shall evaluate and discuss the lessons I learnt during this coursework, the limitations of my code and what I will do in the future accordingly.

In terms of the lessons I learnt, I would say it is fair to say that the mathematics involved in the creation of graphics and rendering light are very complex. I originally intended to use ray casters for collision detection before I found the simpler box method, but ray casters used complicated maths which may have reduced performance. As well as this, I am more aware of using in-built methods for functionality as opposed to spending considerable time and effort implementing it myself. As well as this, I discovered that it is quite easy to learn new programming skills and modules when the documentation is clear and well-structured. We went into the coursework with no knowledge of three.js or JavaScript or html, so I found myself referring to the three.js documentation regularly, and it was surprisingly helpful.

Secondly, the limitations I found of my implementation, is that there are many keypresses involved and no way for a new user to figure out their uses without trial and error, so I may have to figure out how to include instructions in a three.js web application. As well as this, the script is limited in the texture it loads and how it places them on the cube, the same goes for what model it loads. Perhaps in the future I could make it a more generic application suited for a wider variety of purposes.

In the future, I will make sure I refer more closely to documentation when I encounter difficulties when implementing my code. As well as this, I know to look for existing models and in-built functions to help achieve a wider variety and greater quality of applications that I may develop.