CM20219

Viewing and analysing 3d models using webgl

Alfie Newman

2018

1 **Draw a simple cube**

For this task, a cube must be drawn onto the screen, it must be centred at the origin with the opposite corner points (–1, –1, –1) and (1, 1, 1), faces orthogonal to x-, y-, z-axes. Orthogonal means that they are at right angles. For the cube to be the right size, I created the geometry using a BoxBufferGeometry which I passed a width, height and depth of 2. The cube is automatically centred at the origin, hence the corner points are 1 unit away from the coordinate axis.

The function createCube() is called from the init() function which is run once at the beginning. The cube mesh is stored as a global variable and can be called from any function in the program. The below code snippet includes loading the texture mapping for requirement 7, for requirement one the material simply set the color to white.



Because the cube is shown by default when you open the application, it is added to the scene in the init() function.



2 **Draw coordinate system axes**

For this task, three orthogonal lines (lines at right angles to one another) are drawn to represent the x, y and z axis of the coordinate system. They must then be coloured red, green and blue. In order to do this, I set up a material for the lines with the correct colour, then used the .push() method to add vertices to the line. The vertices added were at values of 10 and -10 on the axis the line was representing, and all other values were 0. Finally, I created the mesh and added it to the scene.

This is executed in the init() function as the lines will be present throughout the running of the program. The line meshes are local variable as they only need to be stored while they are added to the scene, after that they are not required.

var material = new THREE.LineBasicMaterial({

color: 0xff0000

});

var geometry = new THREE.Geometry();

geometry.vertices.push(new THREE.Vector3(-10, 0, 0));

geometry.vertices.push(new THREE.Vector3(10, 0, 0));

var line = new THREE.Line(geometry, material);

scene.add(line);

3 **Rotate the cube**

For this task, the cube must rotate about the x, y and z axes but the axes and camera must remain fixed. To achieve this, I created a function roatate(o) which can be passed any object and will apply a rotation to it on all three axes. I selected 0.02 as a speed, this is the amount you add to the rotation property of the object on each axis with each function call.

function rotate(o) {

var SPEED = 0.02;

o.rotation.x += SPEED;

o.rotation.y += SPEED;

o.rotation.z += SPEED;

}

In order to make the rotation run continuously, I call the rotation function with the desired objects from within the animate function which runs on a loop while the program is running. To allow the user to turn the rotation on and off, I created a global variable shouldRotate. The value of this Boolean is reversed when you press ‘R’ and starts as false as default. This is checked using an if statement on each animate loop, and if true the correct objects are rotated. I also added a further two Boolean variables to check that that the bunny objects have been added before trying to rotate them as they are not created in init().

if (shouldRotate) {

rotate(cube);

rotate(edgesCube);

rotate(verticesCube);

rotate(bunny);

if(rotateEdgesBunny) rotate(edgesBunny);

if(rotateVerticesBunny) rotate(verticesBunny);

}

case 82: //r = rotate

shouldRotate = !shouldRotate;

break;

4 **Different render modes**

For this task, the cube must be rendered in different modes, specifically:

* Vertex render mode: Shows the 8 vertices of the cube.
* Edges render mode: Shows edges of primitives.
* Face render mode: Shows the 6 faces of the cube.

To show the vertex render mode, I created a new geometry which I pushed the coordinates of each vertex. I also created a new PointsMaterial, with the colour property set to white and the size set to 0.1 to ensure the vertices were clearly visible. I then created the mesh from the geometry and material. This was stored as a global variable, so it can be accessed from any other function.

function createVerticesCube() {

var verticesGeometry = new THREE.Geometry();

var newMaterial = new THREE.PointsMaterial({

color: 0xffffff,

size: 0.1

});

verticesGeometry.vertices.push(

new THREE.Vector3(1, 1, 1),

new THREE.Vector3(1, 1, -1),

new THREE.Vector3(1, -1, -1),

new THREE.Vector3(-1, -1, -1),

new THREE.Vector3(-1, 1, 1),

new THREE.Vector3(-1, -1, 1),

new THREE.Vector3(1, -1, 1),

new THREE.Vector3(-1, 1, -1)

);

verticesCube = new THREE.Points(verticesGeometry, newMaterial)

}

To show the edges render mode, I simply created a new EdgesGeometry, passing it the geometry of the cube from task 1. I then created the edgesCube using LineSegments.

function createEdgesCube() {

var edgesGeometry = new THREE.EdgesGeometry(cube.geometry);

edgesCube = new THREE.LineSegments(edgesGeometry);

}

To show the faces render mode, I used the code shown in task 1 to create the original cube.

The render mode can be selected by the user using the keyboard. It defaults to the face render mode, but the edges and vertices modes can be shown using the ‘E’ and ‘V’ keys respectively. The following code snippet shows how the program handles an ‘E’ keypress.

case 69: // e = edge

scene.remove(cube);

scene.remove(verticesCube);

scene.add(edgesCube);

break;

5 **Translate the camera**

For this task, the user must be able to translate the camera on its left/right vectors, up/down vectors and forward/back vectors. The translation must not be on the axes of the global coordinate system.

To do this, I created a translateCameraOnAxis function which takes an axis vector (e.g. (1,0,0) for x) and an amount by which to translate the camera as parameters, and then translates it.

Firstly the camera’s local rotation is stored as a quaternion:

var cameraRotation = camera.quaternion;

Next the quaternion is applied to the axis vector:

axisVector.applyQuaternion(cameraRotation);

The resulting vector is added to the camera’s position:

camera.position.add(axisVector.multiplyScalar(amount));

The translation function is called on keyboard presses, with the amount set to 0.5.

The following code snippet shows how the program handles an ‘A’ keypress:

case 65: //a = move left

translateCameraOnAxis(new THREE.Vector3(1, 0, 0), -0.5);

break;

6 **Orbit the camera**

For this task, the camera must be orbited around the cube in arc ball mode. This means it must move around the camera look-at point at a fixed distance. It should be able to do this in both latitude and longitude directions.

I created separate functions for left, right, up and down orbits.

7 **Texture mapping**

For this task, a different texture must be applied to each face of the cube without any skew.

To do this, I created a function loadCubeTextures() which creates a new texture loader, then uses this to load 6 images into an array of materials which is returned to be mapped to the cube.

function loadCubeTextures() {

var textureLoader = new THREE.TextureLoader();

var materials = [

new THREE.MeshBasicMaterial({

map: textureLoader.load(`Arsenal\_Images/arsenal1.png`)

}),

new THREE.MeshBasicMaterial({

map: textureLoader.load(`Arsenal\_Images/arsenal2.png`)

}),

new THREE.MeshBasicMaterial({

map: textureLoader.load(`Arsenal\_Images/arsenal3.png`)

}),

new THREE.MeshBasicMaterial({

map: textureLoader.load(`Arsenal\_Images/arsenal4.jpeg`)

}),

new THREE.MeshBasicMaterial({

map: textureLoader.load(`Arsenal\_Images/arsenal5.jpeg`)

}),

new THREE.MeshBasicMaterial({

map: textureLoader.load(`Arsenal\_Images/arsenal6.jpeg`)

}),

];

return materials;

}

To make the directories manageable, I put all of the images into a subdirectory. I also cropped all images to be 1:1 aspect ratio and powers of two.

The cube geometry created the same way as in task 1, however the material is loaded using the array of cube textures as a new MeshLambertMaterial as follows:

var material = (true) ? loadCubeTextures() : new THREE.MeshLambertMaterial();

As the faces view is default, the textures are loaded and the cube is added to the scene in init(). It can also be shown by pressing ‘F’.

8 **Load a mesh model from .obj**

For this task, a mesh model must be loaded and displayed, before being correctly scaled and translated to fit inside the cube.

To load the mesh model, I created an OBJLoader. Calling it’s load() method, I passed it the path of the .obj file and a function to process the object. Within this function I scaled the object and assigned it to a global variable bunny so it can be accessed from all functions.

function createBunny() {

var bunnyLoad = new THREE.OBJLoader();

bunnyLoad.load(

'bunny-5000.obj',

function (object) {

object.scale.set(0.4, 0.4, 0.4);

object.position.x -= 0.5;

object.position.y -= 0.5;

object.position.z -= 0.5;

bunny = object;

}

)

}

Scaling will multiply all vertices on by the amount specified on each axis. The position is also adjusted by subtracting an amount from the current position of the object on each axis.

The bunny is added to the scene on a ‘B’ keypress, and any other bunny objects are removed as only one render mode should be shown at any time.

case 66: //b = show faces bunny

scene.remove(edgesBunny);

scene.remove(verticesBunny);

scene.add(bunny);

break;

9 **Rotate the mesh, render it in different modes**

For this task, the loaded model (bunny) must be rotated about the x, y or z axis. The bunny should also be shown in vertex, edge and face rendering modes.

To rotate the bunny, I used the same rotation function detailed in task 3. To show the edge render mode, I used the same geometry as the mode imported in task 8 but changed the material such that the wireframe property was set to true. I then create the new mesh as a global variable. Finally, I scale the new mesh the same way I scaled the bunny in task 8. The code snippet for creating the edges bunny is as follows:

function createEdgesBunny() {

if (!rotateEdgesBunny) rotateEdgesBunny = true;

var bunGeom = bunny.children[0].geometry;

var bunMat = new THREE.MeshBasicMaterial({

color: 0xffffff,

wireframe: true

});

edgesBunny = new THREE.Mesh(bunGeom, bunMat);

edgesBunny.scale.set(0.4, 0.4, 0.4);

edgesBunny.position.x -= 0.5;

edgesBunny.position.y -= 0.5;

edgesBunny.position.z -= 0.5;

}

In order to show the vertices rendering mode, I used identical code to that shown above for the edge rendering mode, however I used a PointsMaterial and also Points for the material and mesh respectively.

var pointMaterial = new THREE.PointsMaterial({

color: 0xffffff,

size: 0.02

});

verticesBunny = new THREE.Points(bunnyGeometry, pointMaterial);

10 **Be creative – do something cool!**

In order to go beyond the specification, I decided to implement a prototype of a simple football game using three.js. I created a pitch, football and goal posts using textures and mesh objects. The user can choose a direction for the football both vertically and horizontally using the arrow keys, and when they press shoot the ball will move in this direction until it reaches the goal line. The game will then decide whether or not they have scored based on whether the ball is within the bounds of the goal posts. Other features I have implemented include:

* Input a name to customise instructions.
* Keep score, display victory message when score of three is reached and result the score.
* Ball spins while it moves towards the goal.
* Ball returns to start point (“penalty spot”) after each shot.