Fundamentals of Visual Computing - three.js

Alex Dawkins

December 2020

Part I
Requirements

1 Draw a simple cube

1.1 Explanation

This requirement is simply to draw a cube, with opposite vertices at `(1, 1, 1), (-1 ,-1 ,-1)`, meaning the cube must be 2 metres wide. This is achieved by creating a `geometry` and a `material`, and then using these to create a `mesh`. This is very simple to do in three.js, and might be considered similar to a "Hello, World!" demonstration: used to show a new user how to get started in a new language.

1.2 Implementation

Due to the popularity and simplicity of this requirement, it was very easy to find how to do this in the three.js documentation [2]. This made it a pleasant first milestone for me, as I haven't worked with three.js or WebGL before.

The following code produces a red cube centred at `(0,0,0)` with width, height and depth of 2 metres:

```
const geometry = new THREE.BoxGeometry( 2, 2, 2 );
const material = new THREE.MeshBasicMaterial({color: 0xff0000});
cube = new THREE.Mesh( geometry, material );
scene.add(cube);
```

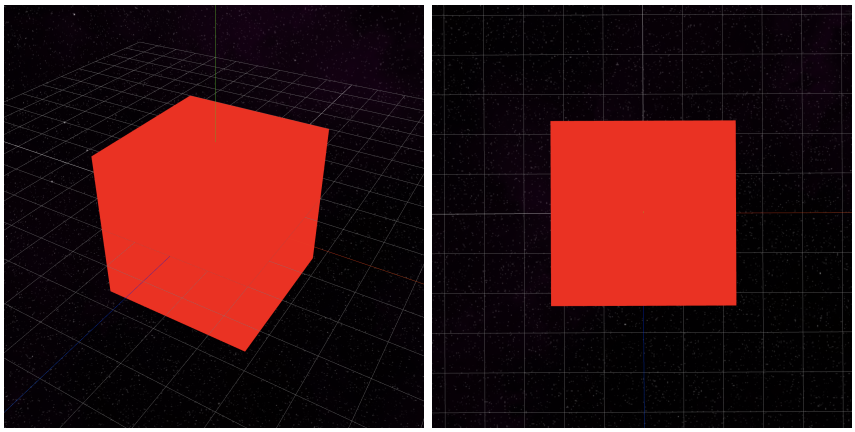
Almost every line of the above code references three.js, showing how much of the actual work the library does for the developer. This requirement simply needs the passing of the correct parameters, and a basic understanding of instantiating objects in a three.js scene.

1.3 Evaluation

The given code produces the results seen in figures 1a and 1b.

Although it's hard to see where each face meets the next one, you can clearly see the general shape of the cube in figure 1a.

As the camera being used is a *Perspective Camera*, objects appear larger the closer they are to the camera, hence the fact that the cube looks slightly larger than width and depth of 2m in figure 1b. However, the grid behind the cube clearly shows that the cube is indeed the correct size (the grid has divisions of 0.5 metres).



(a) Default **(b)** Top

Figure 1 Views of the cube

2 Draw coordinate system axes

2.1 Explanation

The next requirement is to draw axes for each of the three dimensions (x, y, z). This is a common addition to have in 3-dimensional applications, as it is often disorientating and easy to get lost for users if there aren't sufficient guides and indicators of their current location and orientation.

2.2 Implementation

Three.js has again provided a great developer experience, allowing the the use of an *AxesHelper* to visualise each dimension:

```
const axesHelper = new THREE.AxesHelper( 5 );
scene.add( axesHelper );
```

The 5 given as a parameter to the *AxesHelper* constructor indicates the length of each indicative line. In this case, a length of 5 metres will be used.

2.3 Evaluation

In figure 2, you can see the axes helper's individual lines for each dimension. These lines will always point in the correct direction (positive x, y, and z) no matter where the camera is.

3 Rotate the cube

3.1 Explanation

This requirement is for the cube to be rotated about each of the 3 axes independently. Mathematically, this is quite simple to do, as the rotations will be occurring around the origin (0,0,0), so translating before and after the rotation is unnecessary (as it would if the rotation was about another, arbitrary point).

The complications for this requirement lay mainly in the controlling of the rotation. I used the keys `[X]`, `[Y]`, and `[Z]` to allow the user to toggle rotation about the respective axes.

3.2 Implementation

`rotate` is an object containing boolean values for each dimension, stored under the respective key:

```
let rotate = {
    x: false,
    y: false,
    z: false
};
```

This is the key-down event handler, which handles the pressing of the rotation keys, along with others. Each value is toggled each time the key is pressed.

```
function handleKeyDown(event)
{
    switch (event.keyCode)
    {
        /*...*/

        // Cube Rotation
        case 88: // x = rotate cube in X
            rotate.x = !rotate.x;
            break;
        case 89: // y = rotate cube in y
            rotate.y = !rotate.y;
            break;
        case 90: // z = rotate cube in z
            rotate.z = !rotate.z;
            break;

        /*...*/
    }
}
```

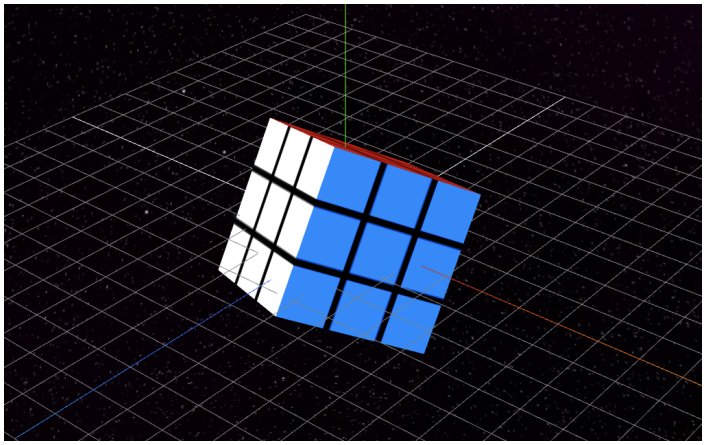


Figure 2 The cube after rotation (textures applied from requirement 7 to better show each face)

The actual rotation is performed in the `animate` function. It works by iterating through each value in `rotate` (and checking it's not been inherited [4]). `rotationSpeed` is a global constant, controlling the the rotational speed in all axes.

```
function animate()
{
    requestAnimationFrame(animate);

    // TO DO: This is a good place for code that rotates your cube (requirement 3).
    for (const dimension in rotate){
        if (rotate.hasOwnProperty(dimension) && rotate[dimension] === true) {
            cube.rotation[dimension] -= rotationSpeed;
            points.rotation[dimension] -= rotationSpeed;
            line.rotation[dimension] -= rotationSpeed;
        }
    }

    /*...*/

    // Render the current scene to the screen.
    renderer.render(scene, camera);
}
```

Again, three.js is doing most of the hard work for us here, as it is handling the rotation mathematics, leaving only the rate of rotation to the developer.

3.3 Evaluation

It is evident in figure 2 that the cube has been rotated around the origin, and that the camera and axes have remained stationery, as per the requirement specification.

4 Different render modes

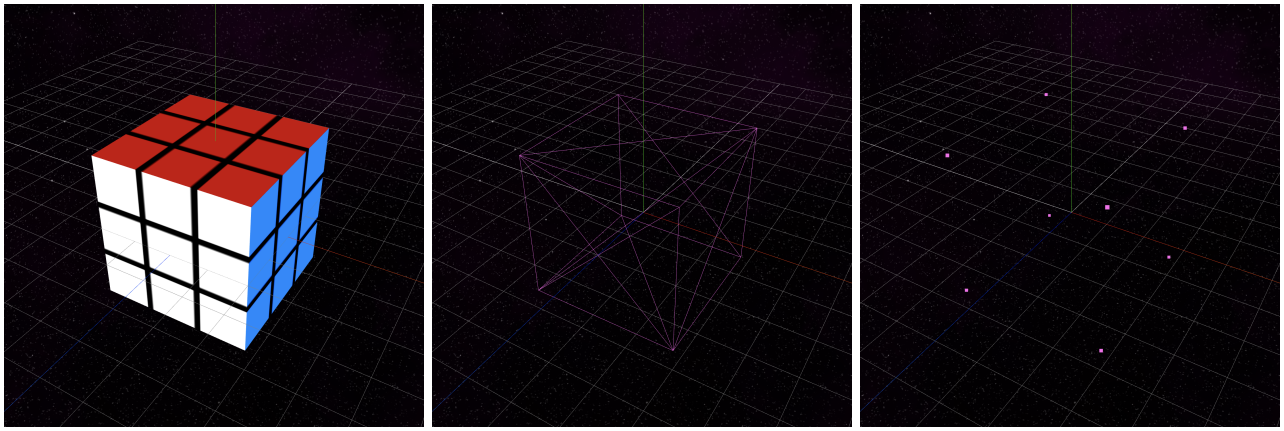
4.1 Explanation

This requirement requires the cube to be rendered in three different ways:

1. **Vertex** - Display only the vertices of the cube. Mapped to `[V]`.
2. **Edge** - Display only the edges of the cube. Mapped to `[E]`.
3. **Face** - Display the faces of the cube (at most 3, of course). Mapped to `[F]`.

4.2 Implementation

Each "render mode" is actually it's own object, built from the same geometry as the others - shown below. `line` is for edge render mode, `cube`, is for face render mode, and `points` is for vertices render mode.



(a) Face **(b)** Edge **(c)** Vertex

Figure 3 Render modes

```
const geometry = new THREE.BoxGeometry( 2, 2, 2 );
line = new THREE.Line(geometry,
    new THREE.PointsMaterial({color: 0xff66ed, size: 0.2})
);
cube = new THREE.Mesh( geometry, materials);
points = new THREE.Points(geometry,
    new THREE.PointsMaterial({color: 0xff66ed, size: 0.1})
);
```

Similar to requirement 3, the key-down events are captured to allow the user to control the rendering mode. The following is in the event handler, seen previously.

```
/*...*/

case 69: // e = edge
    scene.remove(points);
    scene.remove(cube);
    scene.add(line);
    break;

/*...*/
```

The other two render modes follow the same logic: remove the other two objects from the scene, and then add the relevant one. Checks for the object being in the scene aren't necessary, as three.js already checks this for us - no errors (or warnings) are thrown by this code.

4.3 Evaluation

Figure 3 shows each render mode. Rotations are reflected and visible in each mode.

5 Translate the camera

5.1 Explanation

This requirement requires the camera to be translated on it's local x, y and z axes.

5.2 Implementation

I decided the camera would be moved with the **arrow keys**, and `[Q]`, `[A]` for **up** and **down**, respectively. To achieve this, I created an object called `cameraMove` to keep track of which direction the camera should be moved in, similar to the `rotate` object.

```
let cameraMove = {
    up: false,
    down: false,
    left: false,
    right: false,
```

```
    forward: false,
    backward: false
};
```

The `moveCamera` function applies the appropriate movement, according to `cameraMove`.

```
function moveCamera() {
    if (cameraMove.forward)     camera.translateZ( -cameraSpeed );
    if (cameraMove.backward)    camera.translateZ( cameraSpeed );

    // etc... for each direction
}
```

The values within `cameraMove` are controlled from the key-down and key-up event handlers. If a key is pressed, the relevant direction is set to `true`, and then back to `false` when the key is released.

The keys for moving the camera are the only ones that are checked for in the key-up event handler.

5.3 Evaluation

Please run the code and use the **arrows keys** and `[Q]`, `[A]` to move the camera in all 6 directions.

6 Orbit the camera

6.1 Explanation

This was, in my opinion, the most challenging requirement to fulfil. As the use of the three.js `OrbitControl` library was forbidden, I ended up learning a lot about how this popular camera control technique works. I attempted this in a few ways, but wasn't satisfied until I tried the following.

6.2 Implementation

My implementation works by converting from *cartesian coordinates* to *spherical coordinates*, which represent a point in 3D space by it's distance from a fixed origin, it's angle from the X-axis, and it's angle from the Z-axis [3].

I began with using three.js' built-in `Spherical` class [8], but decided to replace it with the fundamental mathematics after I saw that it worked well.

The following code is most of the code responsible for moving the camera. `deltaX` and `deltaY` are the amounts the mouse has moved since the last update cycle.

```
// Convert from cartesian to spherical
const radius = camera.position.length();
let theta = Math.atan2(x, z);
let phi = Math.acos(Math.min(Math.max(y / radius, -1), 1));

// Make changes
phi += deltaY;
theta += deltaX;

// Set camera position to new values (converted back to Cartesian)
camera.position.set(
    (radius * Math.sin(phi) * Math.sin(theta)),
    (radius * Math.cos(phi)),
    (radius * Math.sin(phi) * Math.cos(theta))
);
```

To allow for a variable look-at point, the code will translate by the *negative* coordinates of the look-at point, run the above code, and then translate back by the *positive* look-at point.

6.3 Evaluation

Please run the code and drag your mouse to view this code working.

7 Texture mapping

7.1 Explanation

Requirement 7 is to map a different texture to each side of the cube. As each side must be unique, I decided to draw Rubik's Cube [6] textures in Adobe Photoshop, and map them to the correct face of the box.

7.2 Implementation

After creating the textures and adding them to my project directory, I created an array of the textures in the right order.

```
const materials = [
    new THREE.MeshPhongMaterial({map: loader.load('res/rubiks/blue.jpg')}),
    new THREE.MeshPhongMaterial({map: loader.load('res/rubiks/green.jpg')}),

    // etc...
];
```

A `MeshPhongMaterial` differs from a `MeshBasicMaterial` as it reacts with light, which I add later on.

The `materials` array is then passed to the `Mesh` constructor instead of the `MeshBasicMaterial` as seen in requirement 1 to add the textures to the cube.

```
cube = new THREE.Mesh( geometry, materials);
```

7.3 Evaluation

The result can be seen in figure 3a from requirement 4.

8 Load a mesh model from .obj

8.1 Explanation

Requirement 8 is to load the "Stanford Bunny" [9] into the scene.

8.2 Implementation

Through the use of the provided `OBJloader.js`, this was relatively simple. After instantiating a new instance of `OBJloader` called `objLoader`, the following code is run.

```
objLoader.load(
    // url:
    'res/models/bunny-5000.obj',
    function (object) {
        let peterGeometry = object.children[0].geometry;
        // Scale to be within the cube
        peterGeometry.scale(0.48, 0.48, 0.48);
        // Translate the bunny to be within the cube
        peterGeometry.translate(-0.5, 0.01, 0);

        peter = new THREE.Mesh(peterGeometry,
            new THREE.MeshBasicMaterial({ color: 0x14f7ff })
        );
        peter.name = "Peter";

        scene.add(peter);
)
```

Although undoubtedly more complex, it is clear to see the similarities between adding a `.obj` file to a scene, and a simple cube. The object is scaled and translated to fit within the cube, as per the requirement specification.

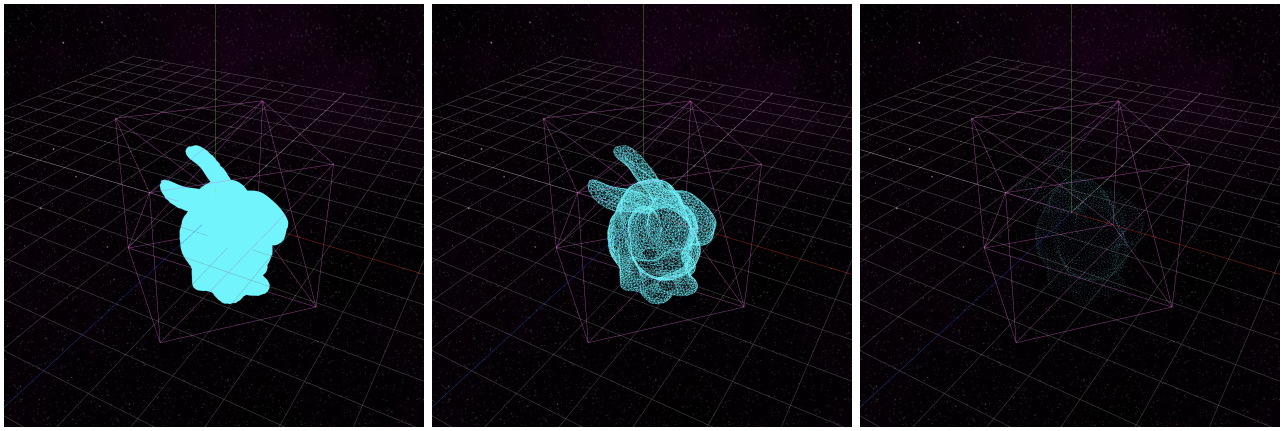
8.3 Evaluation

The "Stanford Bunny" can be seen loaded into the scene in figure 4a.

9 Rotate the mesh, render it in different modes

9.1 Explanation

This requirement is the same as requirements 3 and 4, but for the bunny, instead of the cube.



(a) Mesh **(b)** Edge **(c)** Vertex

Figure 4 Render modes of the "Stanford Bunny", shown to fit within the cube

9.2 Implementation

For vertices and edges mode for the "Stanford Bunny", I added similar code as the for the cube to the above function.

```
// Edges:
peterEdges = new THREE.LineSegments(
    new THREE.EdgesGeometry(peterGeometry),
    new THREE.LineBasicMaterial({ color: 0x14f7ff })
);
// Points:
peterPoints = new THREE.Points(
    peterGeometry,
    new THREE.PointsMaterial({ color: 0x14f7ff, size: 0.01 })
);
```

Both new objects use the same geometry as the normal bunny.

For simplicity's sake, I decided for the keys to control the rotation of the cube would also control the rotation of the rabbit. This not only meant I didn't have to handle different keys being pressed, I could use the same control object: `rotate`.

```
peter.rotation[dimension] += rotationSpeed;
peterPoints.rotation[dimension] += rotationSpeed;
peterEdges.rotation[dimension] += rotationSpeed;
```

9.3 Evaluation

Figure 4 shows the render modes of the bunny in the scene. Note that both the bunny and the cube have been rotated about the y-axis

10 Something creative

This coursework has been extremely enjoyable, and I found great pleasure in adding my own details to the project throughout.

10.1 Skybox

I wanted the cube and bunny to appear as if they were in space, so I decided to add a skybox. A skybox is a way of making the user feel as if they are in an infinite space [7], which is perfect for what I wanted. I found an open-source project [1] for generating a space skybox on GitHub, which I used for the texture.

10.2 Touch Support

After writing `ArcBall.js`, I wanted the project to work on mobile. This was relatively simple, once I had figured out how `touch` events differ from `mouse` events.

10.3 Directional Light

To make the cube slightly more exciting, I added an invisible directional light that orbits the y-axis. This light is coloured white, and has an intensity of 1 (compared to the ambient light's intensity of 0.8). The result is 4 sides of the cube lighting up in sequence, making the cube appear as if it was glowing.

10.4 Stand-alone Camera Orbit Control & Ray-casting

Similar to `OrbitControls.js`, I wanted my camera control code to be able to be used in any three.js project: I wrote it in in a standalone function. This altered the design and development of the function, but I'm happy that it can be reused in any project.

Additionally, I learnt about **ray-casting** by the implementation of my camera control. To find the look-at point to orbit about, a ray is sent from the camera to an invisible x-z plane. This was a good brief experiment into using rays in 3 dimensions. **Scrolling will also dolly the camera towards/away from the look-at point.**

10.5 Messages/Help dialog

To assist the user, I implemented a simple message display feature. This allows for all key commands to be shown to the user, along with other warnings and details. This text can't be selected and doesn't interfere with mouse controls.

Part II
Discussion

11 Lessons learnt

This project has taught me

1. How to work with a 3D graphics framework.
2. How to use spherical coordinates.
3. How many 3D applications I use move their camera (OrbitControl).
4. How to implement a skybox.
5. The basics of ray-casting.
6. How to use LaTeX (due to source code having to be text and syntax highlighted!).

12 Limitations

Due to other deadlines, I didn't have enough time to implement other features I would have liked to.

I also believe JavaScript was a limiting factor, as the fact that it is weakly-typed lead to a lot of bugs throughout development. I attempted to transition to `TypeScript`, though I didn't have sufficient time.

13 Future Development

In addition to improving touch support, I would like to implement some sort of pseudo-gravity, where objects would be attracted to each-other, or maybe just the floor. I would also like to allow the user to create and move cubes within the scene.

I did consider adding *"boids"*: simulated flocking creatures [5]. I have been fascinated by the emergent behaviour that appears in them, and would have liked to program them myself. However, I thought it would be best to save this for a future project, potentially using three.js, but using a more favourable language than JavaScript.

References

[1] Rye Terrell (@wwwtyro). *space-3d*. URL: `https://github.com/wwwtyro/space-3d`.

[2] *BoxGeometry - three.js docs*. URL: `https://threejs.org/docs/#api/en/geometries/BoxGeometry`.

[3] Paul Dawkins. *Section 1-13: Spherical Coordinates*. URL: `https://tutorial.math.lamar.edu/classes/calciii/SphericalCoords.aspx`.

[4] *MDN web docs - Object.prototype.hasOwnProperty()*. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty`.

[5] Craig Reynolds. *Boids*. URL: `https://www.red3d.com/cwr/boids/`.

[6] *Rubik's Cube 3x3*. URL: `https://www.rubiks.com/en-uk/rubik-s-cube-3x3.html`.

[7] *Skybox Basics*. URL: `https://developer.valvesoftware.com/wiki/Skybox_Basics`.

[8] *Spherical - three.js docs*. URL: `https://threejs.org/docs/#api/en/math/Spherical`.

[9] *The Stanford 3D Scanning Repository*. URL: `http://graphics.stanford.edu/data/3Dscanrep/`.