# Lesson 1: An Introduction to *three.js*

Beatriz Sofia Mesquita Gonçalves, 115367

December 2022

**Student** Beatriz Sofia Mesquita Gonçalves, 115367
**Professors** Paulo Dias and Beatriz Sousa Santos

## Introduction

This assignment was developed in the context of the Information Visualization course of the Data Science Master's program at the University of Aveiro. To run the code, create a local server, e.g. live server and run in localhost the pages.
This first delivery consists of a brief introduction to *three.js*. The **outline** of this assignment is:

- Configuration of the environment;
- First example, Visualization pipeline;
- Visualization of a polygonal mesh with color;
- Viewport Update;
- Other primitives in *three.js*.

## 1 Configuration of the environment

We start by setting up the environment. *Three.js* is a library built on webGL to abstract some of the difficulties related to low-level graphics and to reduce the quantity of code to produce the visualizations. Its configuration is similar to the one used by webGL. To use *three.js*, we start by including the following lines in our javascript code:

```
<script src="https://threejs.org/build/three.js"></script>
```

Listing 1: Configuration of the environment

so that we don't have to install *three.js*.

## 2 First example, Visualization pipeline

We create our first *three.js* example based on the tutorial available at `https://threejs.org/docs/index.html#manual/introduction/Creating-a-scene`. This way, we define an HTML to display it:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My first three.js app</title>
    <style>
      body { margin: 0; }
    </style>
  </head>
  <body>
    <script src="https://threejs.org/build/three.js"></script>
    <script>
      // Our Javascript will go here.
    </script>
  </body>
</html>
```

Listing 2: Creating an HTML for display.

So, now all the code below goes into the empty ¡script¿ tag.
We start by defining the **scene**, **camera**, and **renderer**:

```
1
2       // Definition of the scene, camera, and renderer
3       const scene = new THREE.Scene();
4       const camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.innerHeight,
    0.1, 1000 );
5
6       const renderer = new THREE.WebGLRenderer();
7       renderer.setSize( window.innerWidth, window.innerHeight );
8       document.body.appendChild( renderer.domElement );
```

Listing 3: Creating the scene.

There are different cameras in *three.js*, here we use a **PerspectiveCamera**. The first attribute is the field of view, that is, the extent of the scene that is seen on the display at any given moment. The value is in degrees. The second one is the aspect ratio. It is usual to use the width of the element divided by the height, or the image will look distorted. The next attributes are the near and far clipping planes. Objects further away from the camera than the value of far or closer than near won't be rendered. This may be used to get better performance.

For the renderer we used the **WebGLRenderer**, but there are others, often used for older browsers when users don't have WebGL support. Then, we need to set the size at which to render the app. It's a good idea to use the width and height of the browser window.

Now we will define an object/geometry and camera position:

```
1
2       // Definition of an object/geometry and camera position
3       const geometry = new THREE.BoxGeometry( 1, 1, 1 );
4       const material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
5       const cube = new THREE.Mesh( geometry, material );
6       scene.add( cube );
7
8       camera.position.z = 5;
```

Listing 4: Define the object/geometry.

Here we created a cube using a BoxGeometry. This is an object that contains all the vertices and the faces of the cube. In addition to the geometry, it is necessary a material to color. From the several materials, we chose the MeshBasicMaterial, as it is simpler. All materials have properties that will be applied to them.We used a color attribute of 0x00ff00 (green in rgb). A Mesh is also necessary, it basically takes a geometry, and applies a material, which can be inserted into the scene. By default, when scene.add() is called, the thing will be added to the coordinates (0,0,0). This would cause both the camera and the cube to be inside each other. To avoid this, it is necessary to move the camera to z=5.

Now to finish this first example, we render the scene.

```
1
2       // Scene rendering
3       function animate() {
4         requestAnimationFrame( animate );
5
6         // Scene animation
7         cube.rotation.x += 0.01;
8         cube.rotation.y += 0.01;
9
10        renderer.render( scene, camera );
11      };
12
13      animate();
```

Listing 5: Scene rendering.

If we accidentally skip this step, nothing is seen from the HTML file, it it necessary to render the scene. As we can see from the code, we create a loop that causes the renderer to draw the scene every time the screen is refreshed (e.g. 60 fps). Using **requestAnimationFrame** has a number of advantages as it pauses when the user navigates to another browser tab, not wasting processing power and battery life. We can also see that we added to the animated loop a rotation of the cube. Now in Fig.1 we can see a screenshot of the obtained result. The file *firstexample.html* is available in the folder *configurationoftheenvironment*.

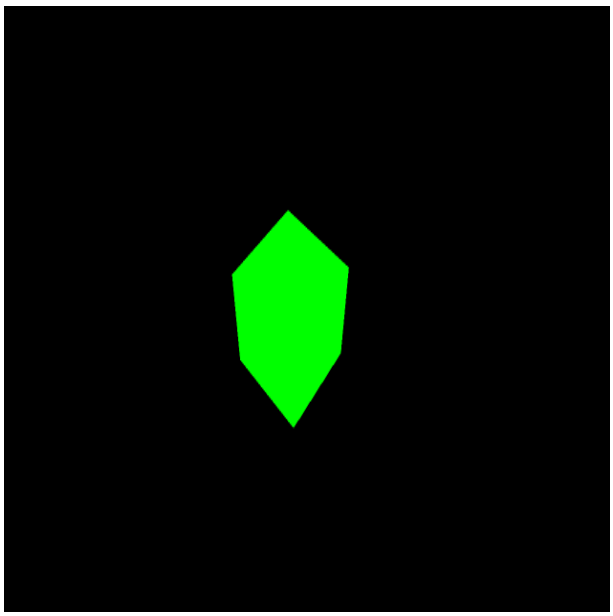

Figure 1: First example: cube rotating.

## 3 Visualization of a polygonal mesh with color

First, we will work on some 2D primitives. We will modify the previous example to visualize a black 2D triangle. Use the following coordinates for the vertices (-1,-1,0) (1,-1,0) and (1,1,0). We add the following code:

```
1
2       // geometry
3           var geometry = new THREE.BufferGeometry();
4
5           // attributes
6           const vertices = new Float32Array( [
7               -1.0, -1.0,  0.0,
8               1.0, -1.0,  0.0,
9               1.0,  1.0,  0.0,
10
11          // 3 vertices per point
12          geometry.setAttribute( 'position', new THREE.BufferAttribute( vertices, 3 ) );
13          ] );
```

Listing 6: 2D Primitives.

Here we defined the geometry, as well as the position of the vertices of our triangle. What we want to do next is to change the color of the triangle, as well as the color of the background. To do this we alter the **material** constant to change the color of the triangle to black, with the color attribute *0x11111*. To add the background color red, we add the function **.setClearColor***(0xb20000)* when rendering, with what is inside the parentheses being the color attribute for the chosen red. Everything else remains the same as in the previous example, as we can see here:

```
1           // geometry
2           var geometry = new THREE.BufferGeometry();
3
4           // attributes
5           const vertices = new Float32Array( [
6               -1.0, -1.0,  0.0,
7               1.0, -1.0,  0.0,
8               1.0,  1.0,  0.0,
9           ] );
10
11
12          // 3 vertices per point
13          geometry.setAttribute( 'position', new THREE.BufferAttribute( vertices, 3 ) );
14
15          // material
16          const material = new THREE.MeshBasicMaterial( { color: 0x111111 } );
17
18          // figure
19          const triangle = new THREE.Mesh( geometry, material );
```

```
            scene.add( triangle );

            // set background color
            renderer.setClearColor( 0xb20000);

        function animate() {
          requestAnimationFrame( animate );

          //cube.rotation.x += 0.01;
          //cube.rotation.y += 0.01;

          renderer.render( scene, camera );
        };

        animate();



         camera.position.z = 5;
```

Listing 7: 2D Primitives and color.

We got the expected output as we can see in the screenshot represented in Fig.2.

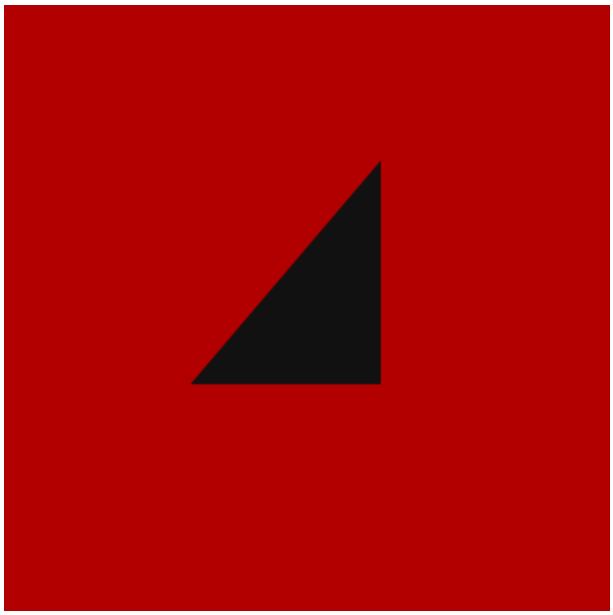

Figure 2: 2D primitives and color.

The file for this output is called *2Dprimitivescolors.html* and can be found on the folder *visualizationofapolygonalmeshwithcolor*.

In the next step, we played a little bit with colors, by mapping different colors in a mesh face, it is necessary to associate a color to each vertice. Besides that, we also change the code for the **material**, as we can see below. The rest of the code remains the same.

```
           // adding color to each vertice
            var colors = new Uint8Array( [
                255,  0,  0,
                0,  255,  0,
                0,  0,  255,
            ] );

            geometry.setAttribute( 'color', new THREE.BufferAttribute( colors, 3, true) );

            // material
            const material = new THREE.MeshBasicMaterial( {vertexColors: true} );

            // figure
            const triangle = new THREE.Mesh( geometry, material );
            scene.add( triangle );

        function animate() {
```

```
        requestAnimationFrame( animate );

        //cube.rotation.x += 0.01;
        //cube.rotation.y += 0.01;

        renderer.render( scene, camera );
      };

      animate();


       camera.position.z = 5;
```

Listing 8: 2D Mapping colors.

Now, we can see the output was the expected, as we can see in Fig.3.

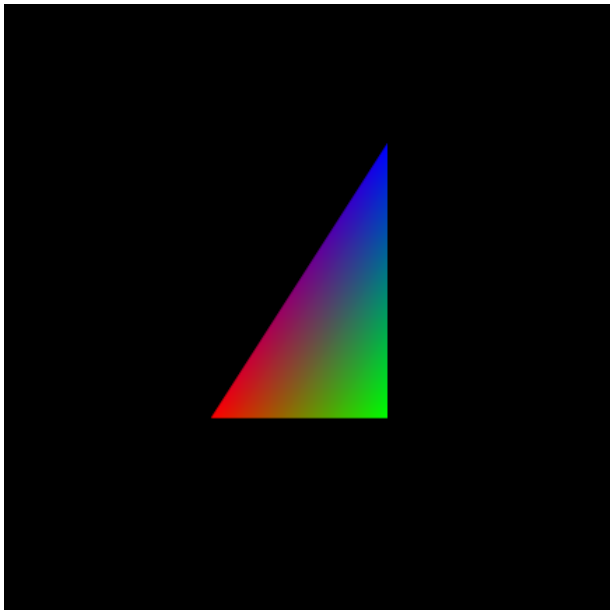

Figure 3: Mapping color.

The file for this output is called *additionofcolor1triangle* and can be found on the same folder as the previous one.

To finish this section, we created multiple triangles, with multiple colors, including the last one we created. To do this, we assigned 4 different geometries for the 4 different triangles we wanted to create:

```
          // geometry
        var geometry1 = new THREE.BufferGeometry();
        var geometry2 = new THREE.BufferGeometry();
        var geometry3 = new THREE.BufferGeometry();
        var geometry4 = new THREE.BufferGeometry();
```

Listing 9: Geometries.

Now, to actually position our triangles we had to assign the position to each one of them and set it to each one of the geometries.

```
          // attributes
    // triang1
        const vertices1 = new Float32Array([
            0.0, 0.0, 0.0,
            0.5, 0.75, 0.0,
            1.0, 0.0, 0.0,
        ]);

    // triang2
        const vertices2 = new Float32Array([
            0.0, 0.0, 0.0,
            -0.35, -1.0, 0.0,
            -0.7, 0.25, 0.0,
        ]);
```

```
15
16      // triang3
17          const vertices3 = new Float32Array([
18              -0.2, 0.15, 0.0,
19              0.35, 0.65, 0.0,
20              -0.85, 0.9, 0.0,
21          ]);
22
23      // triang4
24          const vertices4 = new Float32Array([
25              0.15, -0.95, 0.0,
26              0.90, -0.7, 0.0,
27              0.65, 0.10, 0.0,
28          ]);
29
30
31          // 3 vertices per point
32          geometry1.setAttribute('position', new THREE.BufferAttribute(vertices1, 3));
33          geometry2.setAttribute('position', new THREE.BufferAttribute(vertices2, 3));
34          geometry3.setAttribute('position', new THREE.BufferAttribute(vertices3, 3));
35          geometry4.setAttribute('position', new THREE.BufferAttribute(vertices4, 3));
```

Listing 10: Position.

We add the color just like in the previous example, which is, we assign a color to each vertice of the triangle. For triangles that only had one color, we added the same color to each vertice so it stayed uniform. We add also the colors to the **material**. To be able to see all the triangles, also have to use the "side" flag in the **material** with the argument **THREE.DoubleSide**. This issue is related to the fact that only triangles with normal facing towards the camera are rendered.

```
1           // adding color to each vertice
2          var colors = new Uint8Array([
3              255, 0, 255,
4              255, 0, 255,
5              255, 0, 255,
6          ]);
7          var colors2 = new Uint8Array([
8              255, 255, 0,
9              255, 255, 0,
10             255, 255, 0,
11         ]);
12
13         var colors3 = new Uint8Array([
14             255, 0, 0,
15             0, 255, 0,
16             0, 0, 255,
17         ]);
18         var colors4 = new Uint8Array([
19             0, 0, 0,
20             0, 0, 0,
21             0, 0, 0,
22         ]);
23
24         geometry1.setAttribute('color', new THREE.BufferAttribute(colors, 3, true));
25         geometry2.setAttribute('color', new THREE.BufferAttribute(colors2, 3, true));
26         geometry3.setAttribute('color', new THREE.BufferAttribute(colors3, 3, true));
27         geometry4.setAttribute('color', new THREE.BufferAttribute(colors4, 3, true));
28
29         // material
30         const material1 = new THREE.MeshBasicMaterial({ vertexColors: true, side: THREE.
    DoubleSide });
31         const material2 = new THREE.MeshBasicMaterial({ vertexColors: true, side: THREE.
    DoubleSide });
32         const material3 = new THREE.MeshBasicMaterial({ vertexColors: true, side: THREE.
    DoubleSide });
33
34     // figure
35         const triangle1 = new THREE.Mesh(geometry1, material1);
36         const triangle2 = new THREE.Mesh(geometry2, material2);
37         const triangle3 = new THREE.Mesh(geometry3, material3);
```

Listing 11: Setting color.

Regarding the last triangle, since we want a transparent triangle with a white border, let's define the limits of the triangle and set its color to white (*0xffffffff*), then add it to the scene.

```
1          // define white edges for the 4th triangle
```

```
    const edges = new THREE.EdgesGeometry( geometry4 );
    const line = new THREE.LineSegments( edges, new THREE.LineBasicMaterial( { color: 0xffffff
     } ) );


        scene.add(triangle1);
        scene.add(triangle2);
        scene.add(triangle3);
        scene.add(line);

        function animate() {
            requestAnimationFrame(animate);
            renderer.render(scene, camera);
        };

        animate();


        camera.position.z = 5;
```

Listing 12: Define edges.

So we managed to get a pretty accurate duplicate of the desired image, as we can see in Fig.4

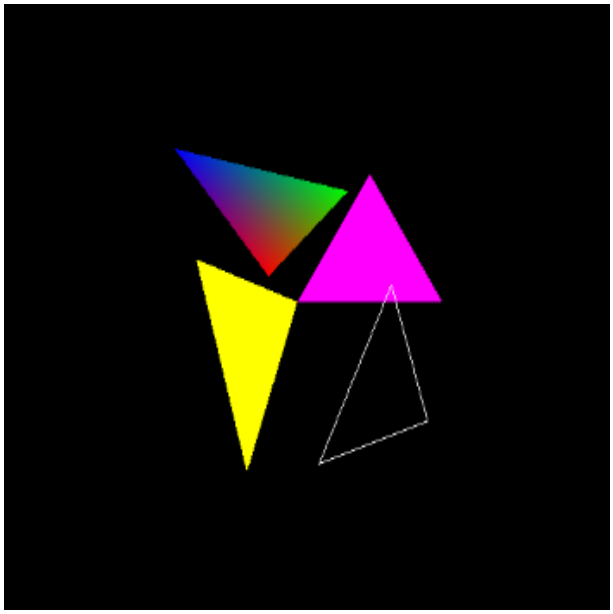

Figure 4: Multiple Polygons.

The file to get this output is called *aditionofcolor4triangles.html* and, again, on the same folder as before.

# 4 Viewport Update

Now, we will go back to the first example (rotation cube). Changing the dimensions of the browser window we can see that the visualization window (viewport) is not updated when the browser window size changes. To solve this problem, we create a new function to be called when the browser window size is updated. This function needs to access the window size (**window.innerWidth** and **window.innerHeight**) and update the renderer size accordingly (**renderer.setSize()**). We also need to modify the aspect camera ratio as well (**camera.aspect=...**) and update this change (**camera.updateProjectionMatrix()**), as we can see in the code bellow:

```
        // viewport_update
            window.addEventListener('resize', function( )
            {
                var width = window.innerWidth;
                var height = window.innerHeight;
                renderer.setSize(width,height);
                camera.aspect = width/height;
                camera.updateProjectionMatrix( );
```

```
9             });
```

Listing 13: Viewport update.

The rest of the code stays the same. So we can see by the following screenshot that the viewport size was shrunk but the image updated to its position:

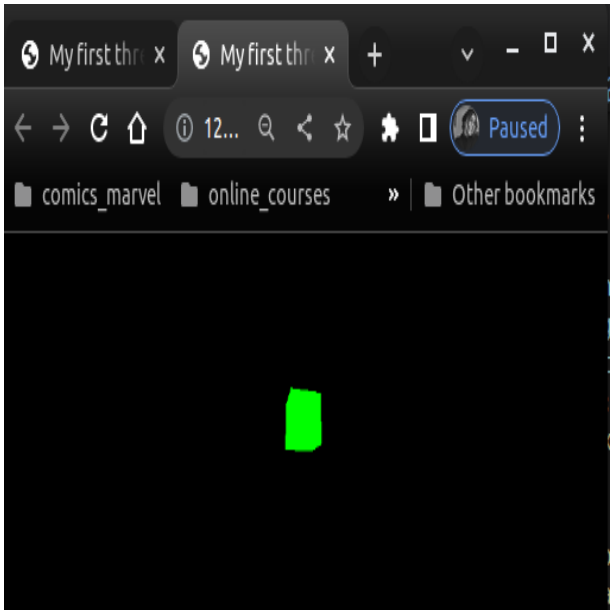


Figure 5: Viewport Update.

This code is in the file *viewportupdatefirstexample.html* and on the folder *viewportupdate*.

## 5 Other primitives in *three.js*

We modified the first example to show the cube in the wireframe. To do that we just have to set the **wideframe** to **true** on the **material**:

```
1       const geometry = new THREE.BoxGeometry( 1, 1, 1 );
2      const material = new THREE.MeshBasicMaterial( { color: 0x00ff00, wireframe: true} );
3      const cube = new THREE.Mesh( geometry, material );
4      scene.add( cube );
5            });
```

Listing 14: Wireframe.

So we can see a screenshot of our output in Fig.6.

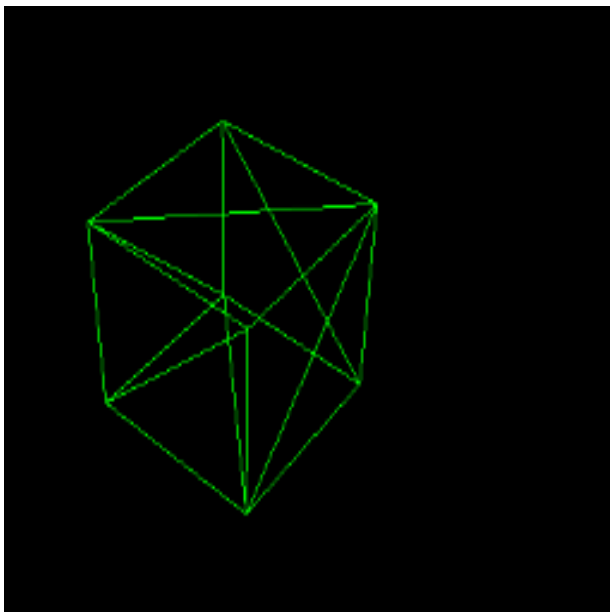

Figure 6: Wireframe.

The code for this is in the file *firstexamplewireframe.html* on the folder *Otherprimitives*.

To finish this assignment, we investigated other available geometries and visualize at least 4 other geometries in the same scene changing some of their default parameters, like size or color. We can see our different outputs here:

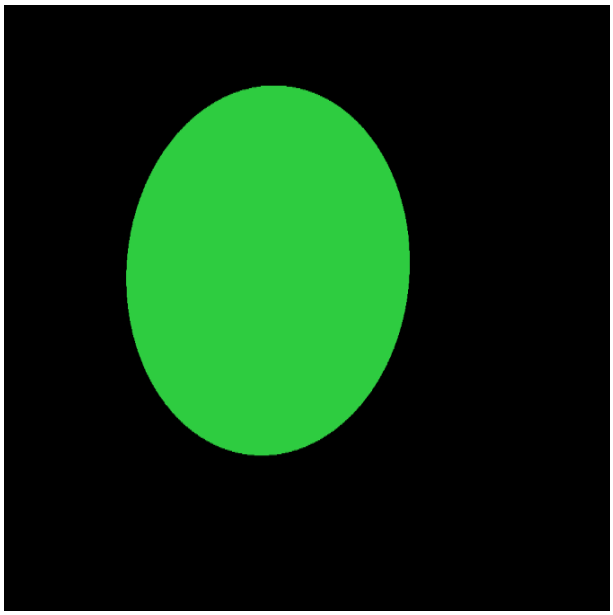

Figure 7: Circle.

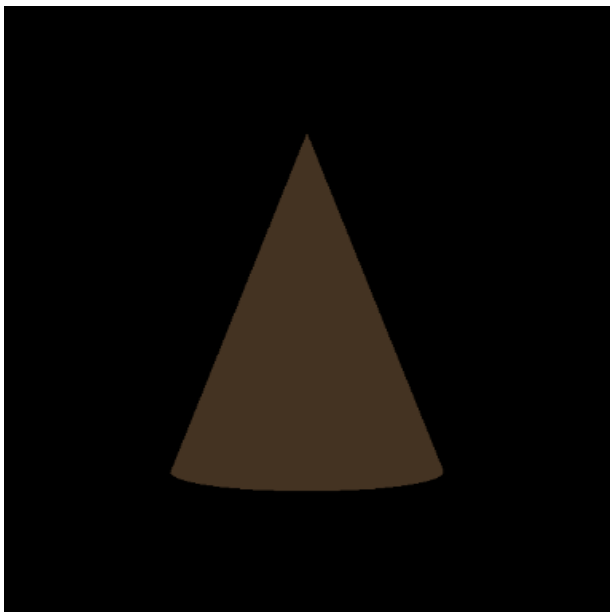

Figure 8: Cone.

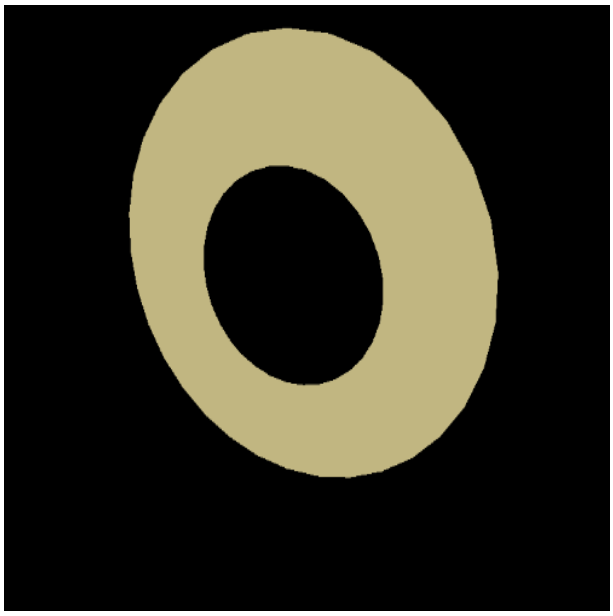

Figure 9: Ring.

The code for these files can be found in the folder *Otherprimitives*.
Now for the last figure, we decided to apply a wireframe again but in a different way:

```
1       const geometry =  new THREE.SphereGeometry( 2, 2, 2 ); // changed default size
2       const material =  new THREE.MeshBasicMaterial( { color: 0xff851b} ); // changed default
    color
3       const sphere = new THREE.Mesh( geometry, material );
4       scene.add( sphere );
5
6             const wireframe = new THREE.WireframeGeometry( geometry );
7
8             const line = new THREE.LineSegments( wireframe );
9             line.material.depthTest = false;
10            line.material.opacity = 1;
11            line.material.transparent = true;
12
13            scene.add( line );
14
15      camera.position.z = 5;
16
```

```
17      function animate() {
18        requestAnimationFrame( animate );
19
20        sphere.rotation.x += 0.01;
21        sphere.rotation.y += 0.01;
22
23        renderer.render( scene, camera );
24      };
25
26      animate();
```

Listing 15: Wireframe.

So, as an output, we get a static wireframe and a rotating figure around this wireframe:

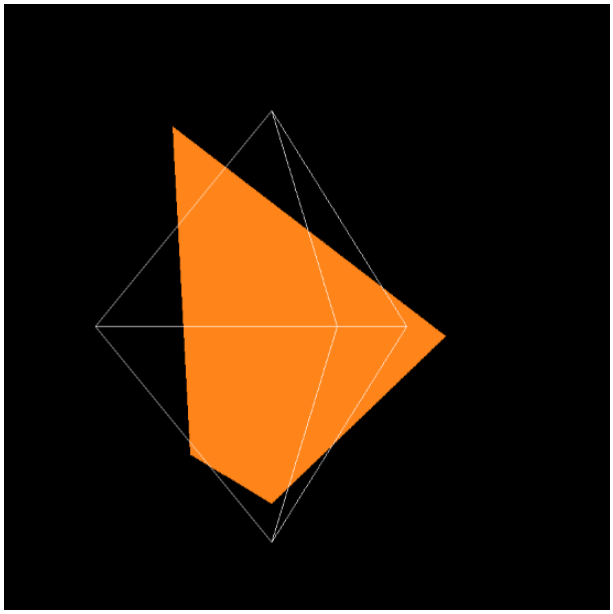

Figure 10: Wireframe2.

The code for this output can be found in the file *WireframeGeometryextra4.html* in the folder *Otherprimitives*.

[3] [1] [2]

# References


[1] three.js examples. `https://threejs.org/examples/`. Accessed: 2022-12-14.

[2] three.js learn. `https://threejs.org/`. Accessed: 2022-12-14.

[3] uainfovis/three.js/lesson01. `https://github.com/pmdjdias/ua_infovis/tree/master/Three.js/Lesson_01`. Accessed: 2022-12-14.