# Three.js - Lesson 1

.

Visualização de Informação

Bruno Bastos 93302
*DETI*
*Universidade de Aveiro*

Leandro Silva 93446
*DETI*
*Universidade de Aveiro*

***Abstract*—Report that shows the resolution of the first Lesson of Three.js of the Information Visualization course. It provides the explanation for the proposed exercises as well as some of the obstacles that appear during the development.**

***Index Terms*—Three.js**

## I. Introduction

Three.js is a javascript library and API used to create 3D Graphics in the Web Browser. It is a high level library, meaning that there is more abstraction, making it easier to program, but making it slower and/or less flexible. It is built on top of WebGL. This report follows the class guide and has examples of how to use and render different objects using the Three.js library.

## II. Initial Cube

In the first example, it is asked to display a 3D cube in the screen and then making it rotate on different axis.

First, in order to display anything on the screen with Three.js, it is necessary to have 3 things: scene, camera and render. The following code is an example on how to create those 3 provided by the Three.js documentation.

```
    const scene = new THREE.Scene();
    const camera = new THREE.PerspectiveCamera( 75,
        window.innerWidth / window.innerHeight, 0.1,
        1000 );

    const renderer = new THREE.WebGLRenderer();
    renderer.setSize( window.innerWidth,
        window.innerHeight );
    renderer.setClearColor(0x000000, 1.0);
    document.body.appendChild( renderer.domElement
        );
}
```

Scenes allow you to set up what and where is to be rendered by Three.js. This is where you place objects, lights and cameras. There are a few parameters that can be adjusted in the scene but for this example it is not necessary to get to know them.

In the code snippet above, the camera used is a *Perpective-Camera*. This camera is designed to mimic the way the human eyes see, which is essentially what we need for our use case. The provided parameters consist of:

- FOV - the field of view of the camera
- Aspect - the camera aspect ratio. Recommended to the canvas width/height.
- Near - near plane. Things won't be displayed if closer than the near plane.
- Far - far plane. Things won't be displayed if further than the far plane.

The renderer uses a WebGL renderer and the setSize function sets the size of the canvas on the screen, taking into consideration the devices display size. The color of the background was also changed using setClearColor. Lastly, the renderer is appended to the HTML as a $< canvas >$ tag.

With these 3 components it is possible to start adding things to see on the screen. The objective of the first exercise is to display a cube and make it rotate along different axis.

```
const geometry = new THREE.BoxGeometry(1,1,1);
const material = new THREE.MeshBasicMaterial( {
    color: 0x00ff00 } );
const cube = new THREE.Mesh( geometry, material );
scene.add( cube );
camera.position.z = 5;
```

The geometry defines the shape of the object, since we want to make a cube, we choose a BoxGeometry by specifying the width, height, depth. The material describes the appearance of the object. Here, we just changed the color to be green. The mesh represents triangular polygon mesh based objects. Finally, the mesh can be added to the scene and the cube will be displayed. Since the camera is in the origin, it is not possible to see the cube, so we moved it a little back by changing its z coordinates. The cube is not rendered yet because we didn't tell the renderer to.

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

        cube.rotation.x += 0.01;
    cube.rotation.y += 0.01;
}
render();
```

We define our render function which will run on loop with 60fps thanks to requestAnimationFrame. By calling the render function from the renderer and providing our scene and camera

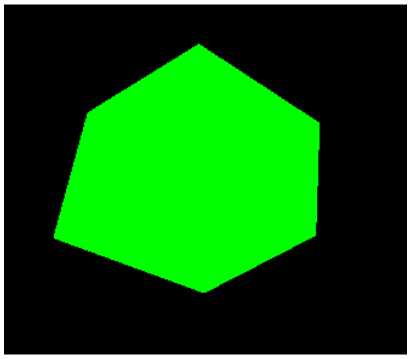

Fig. 1. Resultant initial cube.

we can finally see the cube in the screen. Now, since our render function is called on a loop, we can increase the cube rotation on different axis by a small value, and it will be rendered everytime the render function is called. Figure 1 shows the resultant cube.

## III. Triangles

This exercise objective is to display multiple triangles in the screen, each one in a different position and with a different color.

The only that changes from the cube example is that now we do not need to create the cube mesh but a mesh for the triangle.

```
let geometry = new THREE.BufferGeometry();

const vertices = new Float32Array( [
        -1.0, -1.0,  0.0,
        1.0, -1.0,  0.0,
        1.0,  1.0,  0.0,
] );

geometry.setAttribute( 'position', new
    THREE.BufferAttribute( vertices, 3 ) );
```

First we need the Geometry of the triangle. This time we used a BufferGeometry because it allows to define the shape using the list of vertices.

For the material we have two options according to the exercise. A flat color where the triangle is filled with that color or a gradient where it is necessary to specify the color for each of the vertices of the triangle.

```
if (triangle.colors) {
        material = new THREE.MeshBasicMaterial({
            vertexColors: THREE.VertexColors, side:
            THREE.DoubleSide });
        let colors = new
            Uint8Array(triangle.colors);
        geometry.setAttribute('color', new
            THREE.BufferAttribute(colors, 3, true));
}
else if (triangle.color) {
        material = new THREE.MeshBasicMaterial({
            color: triangle.color, wireframe:
            triangle.outlined, side:
            THREE.DoubleSide });
}
```

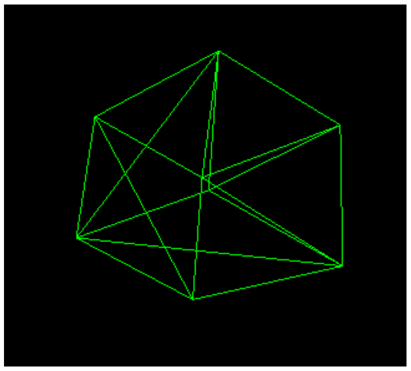

Fig. 2. Resultant initial cube with wireframe.

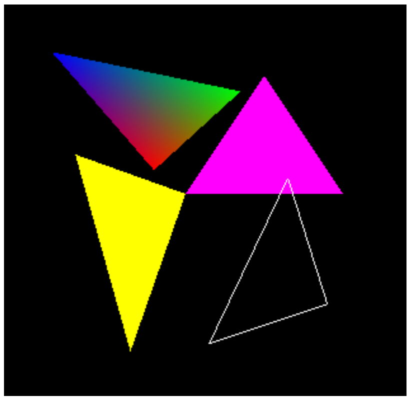

Fig. 3. Resultant four triangles.

In this part of the code, for the triangles with just one color (if below), the material selects the color for the whole triangle. The wireframe is used to only show the borders of the triangle (it is not filled, see figure 2) and the side allows the mesh to be rendered in both sides which in our case was necessary in order to see the triangles (see figure 3).

After that is the same process of creating the mesh and adding it to the scene.

## IV. Viewport Update

One problem with the current work until now is that, when the window resizes, the geometries shown are deviated from the window's center. This happens because the viewport is not updated when the browser window size changes. To solve this, we have to create an event listener that calls a function every time the browser window size is update and update the renderer size accordingly. To do this, we need to access the browser width and size with `window.innerWidth` and `window.innerHeight`, respectively, and update the renderer size with `renderer.setSize()`. This does not completely solve the problem, since we have to update the camera aspect as well. The default aspect ratio of the camera is 1, for a square canvas, therefore we have to change the aspect to something like *canvas width* / *canvas height*. After this, we have to call the function `camera.updateProjectionMatrix()` for the changes to take effect.

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

## V. Other primitives

After having a more deep understanding of how these geometries and materials worked, we created more versions of different geometries and even lines to test it out. For this, we used a cylinder, a cone, an ellipse and a spinal-curve. We also changed its properties according to the elapsed time, which was accessed through the `THREE.Clock()`. Aside from the rotation, the additional properties we changed were the position, scale, color and visibility. Figure 4 shows a frame of our end result.

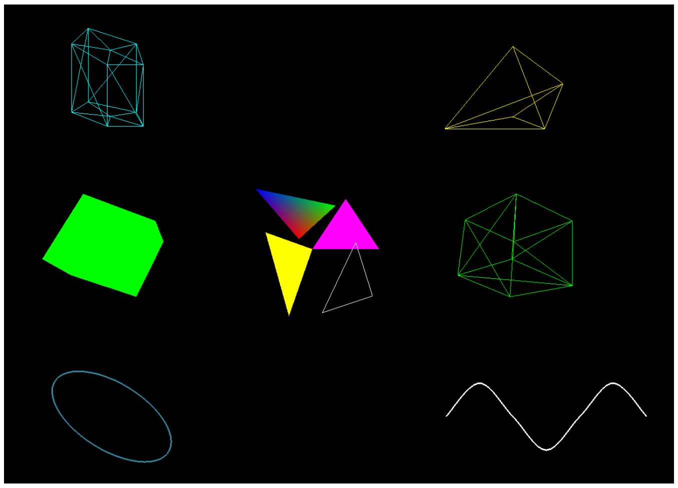
Fig. 4. End result with all geometries and lines.

## VI. Conclusion

The exercises were fully implemented without many difficulties, all due to the well documented Three.js library. Overall, we think that the objective of understanding and getting familiar with the Three.js library was fulfilled.