# 1st Lab work - Three.js

Mariana Rosa 98390, Simão Arrais 85132

Information Visualization, 2022 - Master Degree in Informatic Engineer, University of Aveiro

## Abstract

The present report aims to explain the developed exercises on the first lab work related to Three.js.

## 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.

## First example

In the first example, it is asked to display a 3D cube on screen and then make it rotate on different axes. The code developed in this exercise was taken from the example given by the professor.

To create and display any 3D scene, we first set up a scene, a camera and renderer.

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

// Create Renderer
var renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );
```

Figure 1: Creation of scene, camera and renderer

Scenes allow you to set up what and where the objects will be rendered by Three.js. In this example we make use of the PerspectiveCamera method to mimic the way the human eye sees. The provided parameters to that method are:

- FOV - Camera vertical field of view.
- Aspect - Camera aspect ratio which is recommended to the canvas width and height.
- Near - Camera near plane which doesn't display things that are closer than the near plane.
- Far - Camera far plane which doesn't display things that are further than the near plane.

We also need a *geometry* which defines the shape of the object and since the objective is a cube, we make use of *BoxGeometry* and then specify the width, height and depth. A *material* is also needed to describe the appearance of the object.

```
// Create a cube object and camera positioning
var geometry = new THREE.BoxGeometry(1,1,1);
var material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
var cube = new THREE.Mesh( geometry, material );
scene.add( cube );
camera.position.z = 5;
```

Figure 2: Creation of cube object

To get the final result it's just a matter of adding the cube to the scene and defining a function called *render* that updates the rotation of the cube and the camera's projection matrix.
The function also continuously renders the scene by updating the rotation of the cube using *cube.rotation.x += 0.01* and *cube.rotation.y += 0.01*.

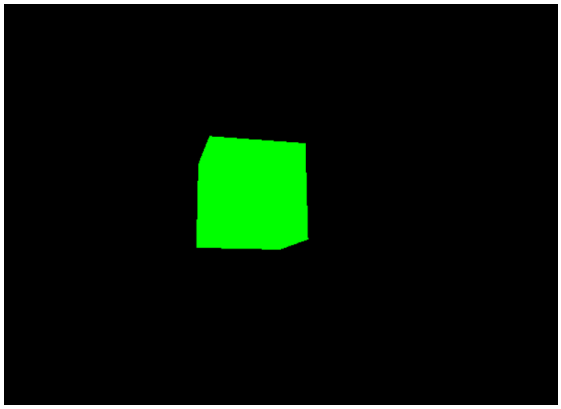

Figure 3: Cube developed for the first exercise

# 2D Primitives

This code creates a simple 3D scene with a black triangle. To create a triangle is basically the same procedure as if it was a cube but in this case, we specify the coordinates of the three vertices using a *Float32Array* object and Three.js *BufferGeometry* method. We then modify the position of the geometry using these coordinates, indicating that the points are in 3 dimensions.

```
var 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 ) );
var colors = new Uint8Array( [
    255,  0,  0,
    0,  255,  0,
    0,  0,  255,
] );
```

Figure 4: Creation of Triangle object

For the material of this figure, we set the color to black and the renderer color to red using the function *setClearColor*, being the result Fig. 5

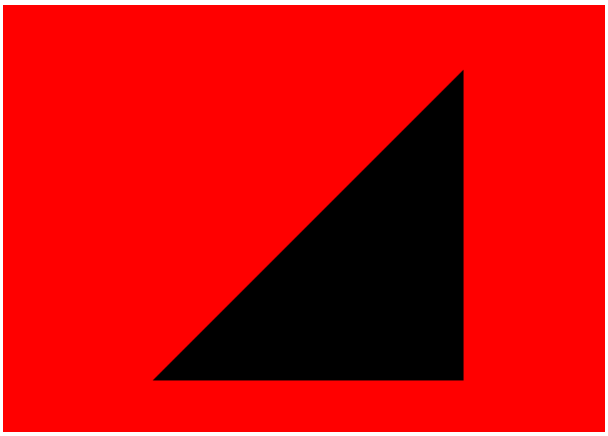

Figure 5: Triangle developed for the second exercise

# 2D Primitives (with color)

```
var colors = new Uint8Array( [
    255,  0,  0,
    0,  255,  0,
    0,  0,  255,
] );
geometry.setAttribute( 'color', new THREE.BufferAttribute( colors, 3, true) );
```

Figure 6: Procedure to add color t

To add color to the triangle primitive, we added color to the 3 vertices, the resulting being this way:

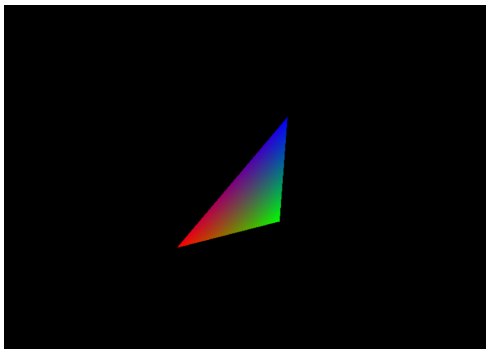

Figure 7: Triangle with color

But the goal of this exercise is to build four triangles. The procedure to get these Triangles is the same as if it was only one, the difference here is the color and the positioning of each and every single triangle. The vertex coordinates for each triangle are set using *Float32Array* and the colors for each triangle are set using *Uint8Array*. The positions and colors for the triangles are set as geometry attributes and the triangles are created as *THREE.Mesh* objects. The script will continually render the scene with the four triangles. Being the final result Figure 8.

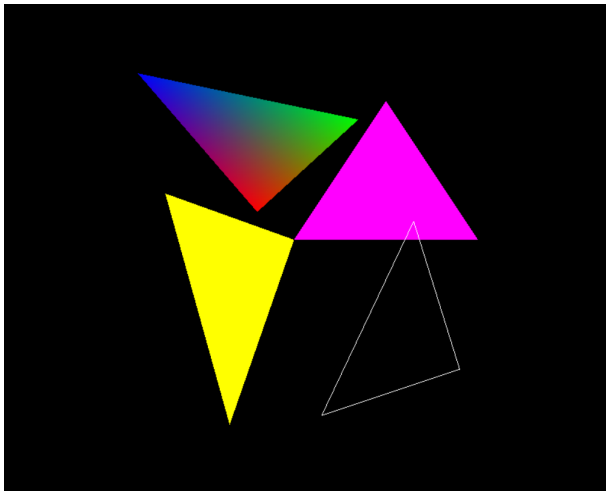

Figure 8: Result of colored triangles

# Viewport update

For this exercise, we were required to modify the initial exercise and update the visualization window so that the cube would still be visible even when the viewport (visualization window) changed. To accomplish this, we added an event listener to the

window object to ensure that the 3D scene is correctly displayed whenever the browser window is resized. This is done just by adding a simple function and an *eventListener*. Without this code, it would look like this:

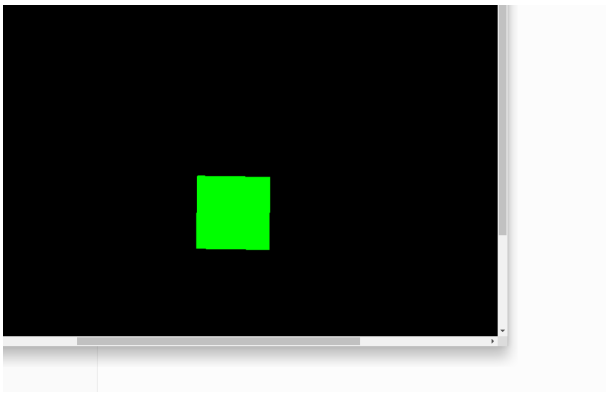

Figure 9: Resized browser without *eventListener*

With those changes, when the user resizes the window, the *rendered* and *camera* size are updated accordingly keeping everything with the same look.

## Other Primitives

After all the exercises we were able to have a more deep understanding of how Three.js and its geometries and materials worked.
We were asked to display the cube on the first exercise as a wireframe and explore other geometries.

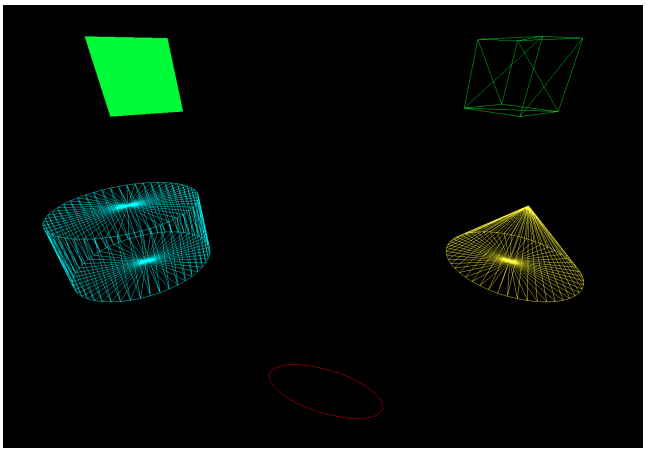

Figure 10: Other geometries

To achieve the cube as a wireframe is as simple as passing the parameter of *wireframe* as *true* in the creation of the *MeshBasicMaterial*.

For the cone, we can think about it as if it was a pyramid with a lot more vertices at the base. It uses the *ConeGeometry* method where the parameters are *radius*, *height, radialSegments* (equivalent to the vertices at the base).

```
// Cone
geometry = new THREE.ConeGeometry(1, 1, 50);
material = new THREE.MeshBasicMaterial({ color: 0xffff00, wireframe: true });
const cone = new THREE.Mesh(geometry, material);
cone.position.set(3, 0, 0);
scene.add(cone);
```

Figure 11: Code for the cone creation

The cylinder uses the same logic as the cone but calls in a different method, which is *CylinderGeometry* and takes *radiusTop, radiusBottom, height, radialSegments*.

```
// Cylinder
geometry = new THREE.CylinderGeometry( 1, 1, 1, 50 );
material = new THREE.MeshBasicMaterial( {color: 0x00ffff, wireframe: true } );
const cylinder = new THREE.Mesh( geometry, material );
cylinder.position.set(-3, 0, 0);
scene.add( cylinder );
```

Figure 12: Code for the cylinder creation

It's even possible to draw lines line ellipses that make use of the method *EllipseCurve* but since an ellipse is not a geometry that behaves like the other ones (cube, cone and cylinder), instead of having the material using *MeshBasicMaterial*, it has one using *LineBasicMaterial* which is a special material for drawing wireframe-style geometries. It's also important to note that an ellipse to be a true ellipse needs to have a lot of points, otherwise we would end up with something like what is shown in Fig 13.

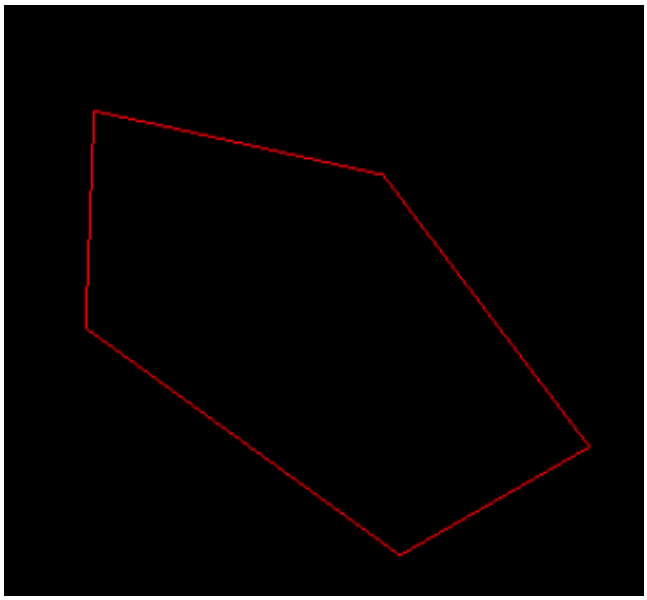

Figure 13: Ellipse using only 5 points

# Conclusion

All the exercises were implemented and overall we think that the objective of understanding and getting familiar with Three.js library was accomplished.

# Links

https://github.com/marianarosa01/guioesVI