Lesson 1 - three.js Introduction

Course: Information Visualization - 44156

André Fernandes (97977) 50.0%, Gonçalo Machado (98359) 50.0%

11/12/2023

Class - TP2

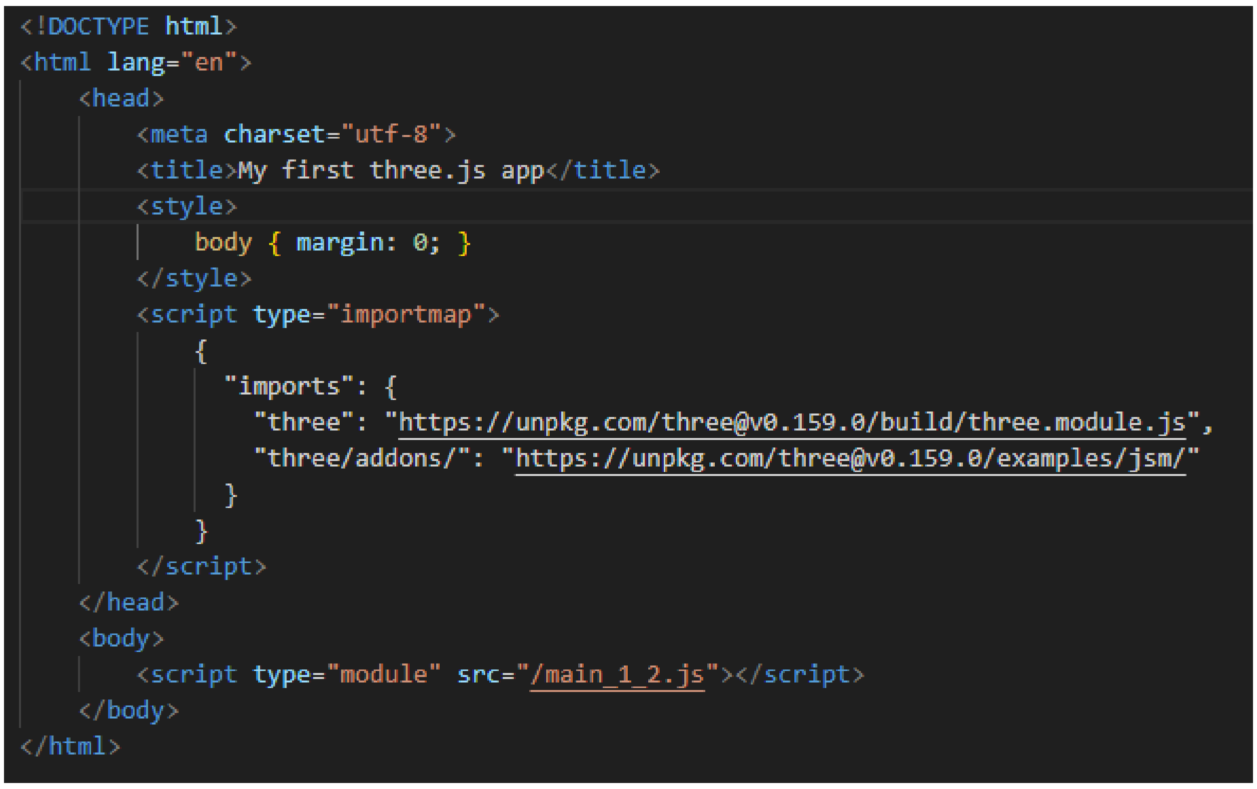
Developed *software* can be found **here**.

Introduction

Three.js, a powerful JavaScript library for creating 3D graphics on the web in the Computer Graphics area. In this report, we will delve into the essential aspects of configuring environments and developing visualization pipelines using Three.js. The primary focus areas covered in this report include the configuration of environment, one first example, 2D primitives, addition of color, view-port update and other primitives.

1.1 - three.js configuration

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, it is necessary to include the lines ilustrated on Figure 1.



```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>My first three.js app</title>
        <style>
            body { margin: 0; }
        </style>
        <script type="importmap">
            {
              "imports": {
                "three": "https://unpkg.com/three@v0.159.0/build/three.module.js",
                "three/addons/": "https://unpkg.com/three@v0.159.0/examples/jsm/"
              }
            }
        </script>
    </head>
    <body>
        <script type="module" src="/main_1_2.js"></script>
    </body>
</html>
```

Figure 1: Our html file to run all the java script files.

In order to make use of the library, it is necessary the following line in each java script file developed:

```
import * as THREE from 'three';
```

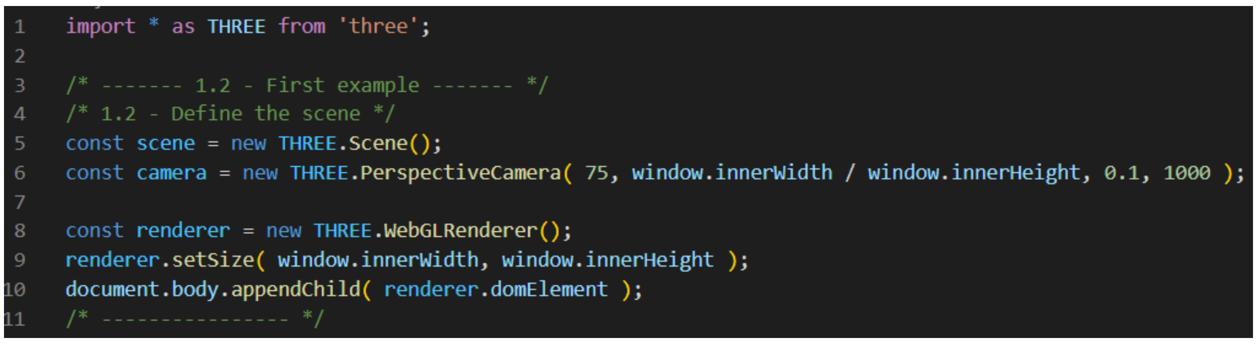
1.2 - First example

For the first example, we had to follow a tutorial available. This tutorial consisted of four steps.

The first step consisted in the definition of the scene, camera and renderer, the second in the definition of an object/geometry and camera position, the third in rendering the scene and the fourth one in the scene animation.

0.1 Definition of the scene, camera and renderer

The code used to make this definitions was the one on Figure 2.



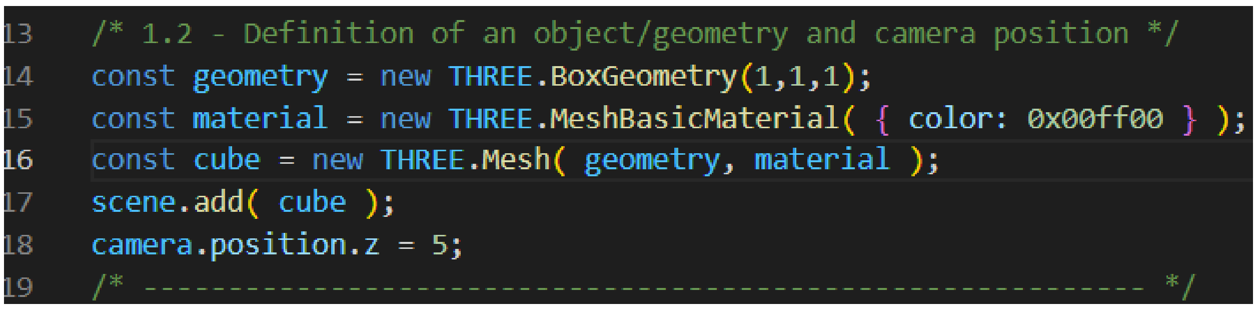
```
import * as THREE from 'three';

/* ------- 1.2 - First example ------- */
/* 1.2 - Define the scene */
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 );
document.body.appendChild( renderer.domElement );
/* --------------- */
```

Figure 2: Code for definition of the scene, camera and renderer of 1.2 - First example.

First is created a scene that works as a container that holds all the objects, lights, and cameras in a 3D environment. Then, it is created a camera that creates a new perspective. This renderer is used to display the 3D scene using WebGL, which is a JavaScript API for rendering interactive 2D and 3D graphics in compatible web browsers. It is set with the window sizes.
The final line appends the renderer to the html main body element.

0.2 Definition of an object/geometry and camera position

The code used to make this definitions was the one on Figure 3.



```
/* 1.2 - Definition of an object/geometry and camera position */
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;
/* ----------------------------------------------------------- */
```

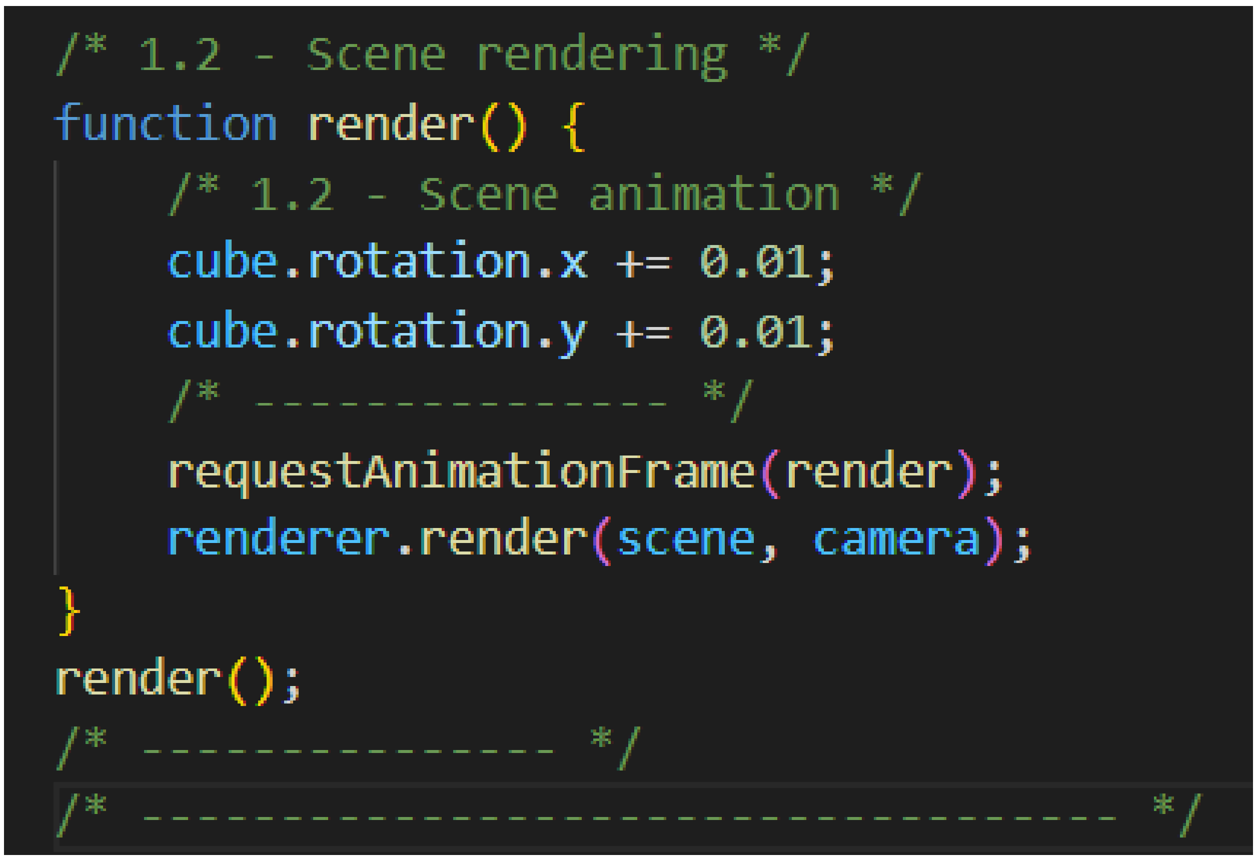
Figure 3: Code for definition of an object/geometry and camera position of 1.2 - First example.

The where a new instance of THREE.BoxGeometry is created represents a cube or rectangular box in three-dimensional space. The parameters 1, 1, 1 define the width, height, and depth of the box, respectively. The instance of THREE.MeshBasicMaterial is a simple material that is not affected by lights. The cube variable holds a 3D cube by putting a

material on a geometry.
The add function adds the created cube mesh to the scene and the camera one sets the position of the camera along the z-axis to 5 units.

0.3 Scene rendering & animation

The code used to make this definitions was the one on Figure 4.



```
/* 1.2 - Scene rendering */
function render() {
    /* 1.2 - Scene animation */
    cube.rotation.x += 0.01;
    cube.rotation.y += 0.01;
    /* --------------- */
    requestAnimationFrame(render);
    renderer.render(scene, camera);
}
render();
/* --------------- */
/* ------------------------------------ */
```

Figure 4: Code for scene rendering & animation of 1.2 - First example.

Here, it is created a function that make an animation by updating the rotation of the cube.

Next, it used a method that tells the browser that we wish to perform an animation and requests that the browser to call a specified function to update an animation; the last line of the function is responsible for rendering the 3D scene.

0.4 Results

In Figure 5 it is represented on frame of the animation obtained.

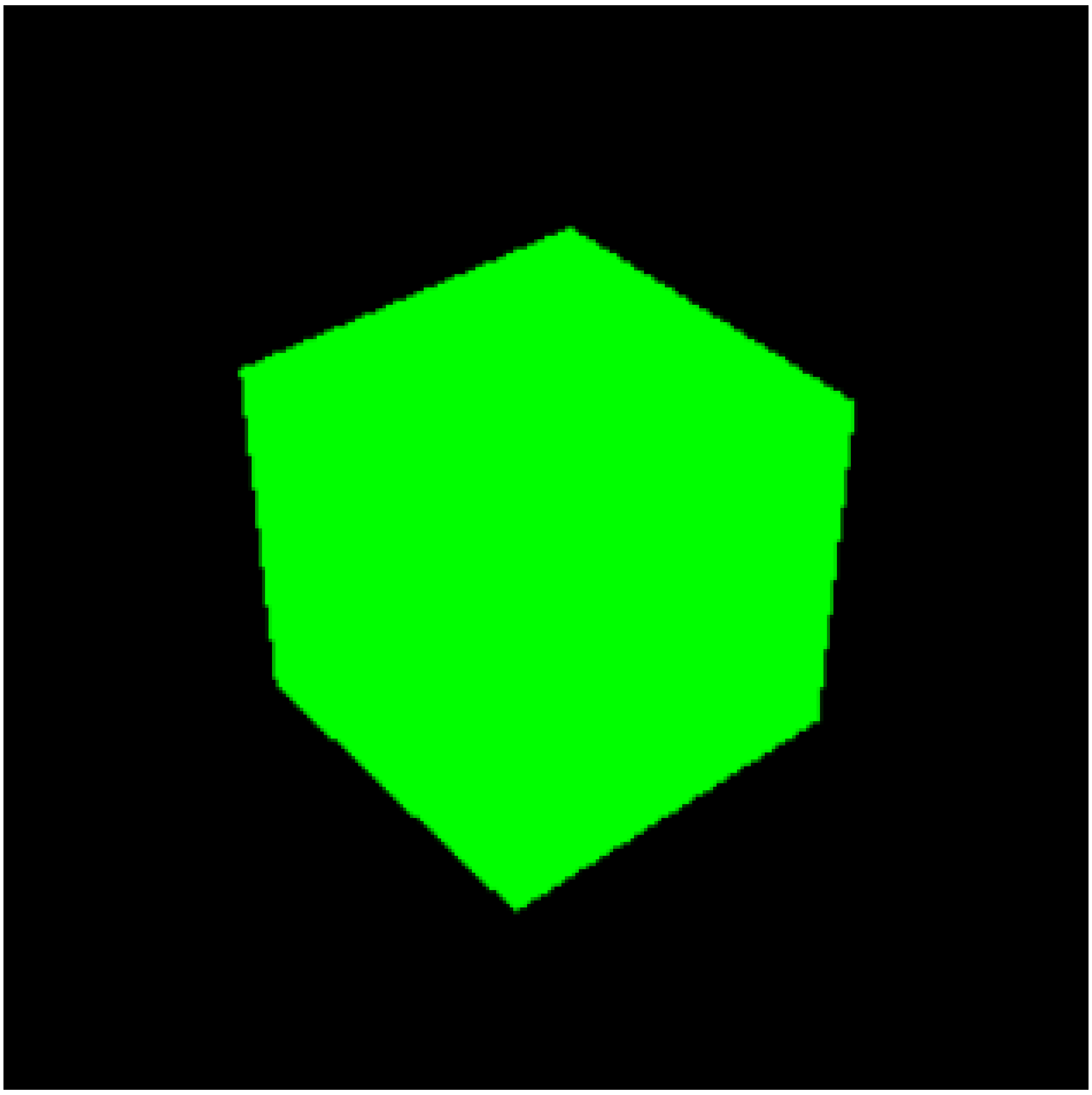


Figure 5: Result of 1.2 - First example.

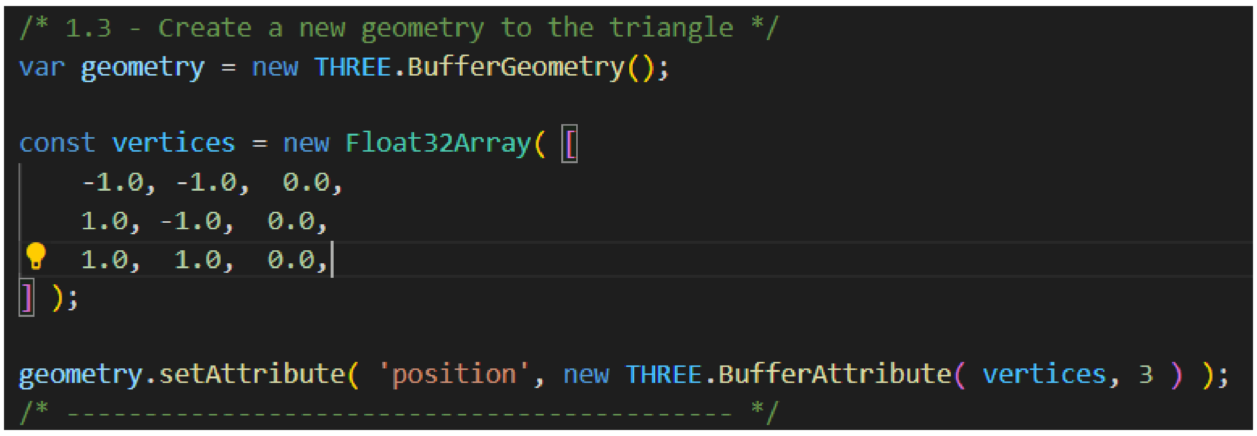
1.3 - 2D primitives

In this exercise we were asked to modify the previous example to visualize a black 2D triangle and to create another material so that the triangle is black over a red background without modifying the camera position.

To obtain a triangle we needed a new geometry so, we commented the previews one and created a new one as a buffer. Then we defined the three vertex of the triangle. Finally, we assign vertex data to the buffer specifying that the attribute is composed of three values per vertex, corresponding to the (x, y, z) coordinates; Figure 6.

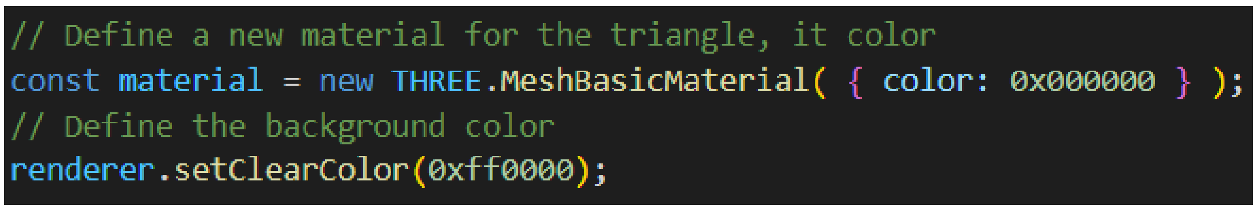
For the new color scheme, we define a new material with a wanted color and, to define the color of the background, we have used the ".setClearColor" function; Figure 7.

The result, a frame of the animation, is represented in Figure 8.



```
/* 1.3 - Create a new geometry to the triangle */
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 ) );
/* ------------------------------------------- */
```

Figure 6: Code for the triangle definition of 1.3 - 2D primitives.



```
// Define a new material for the triangle, it color
const material = new THREE.MeshBasicMaterial( { color: 0x000000 } );
// Define the background color
renderer.setClearColor(0xff0000);
```

Figure 7: Code for the triangle color definition of 1.3 - 2D primitives.

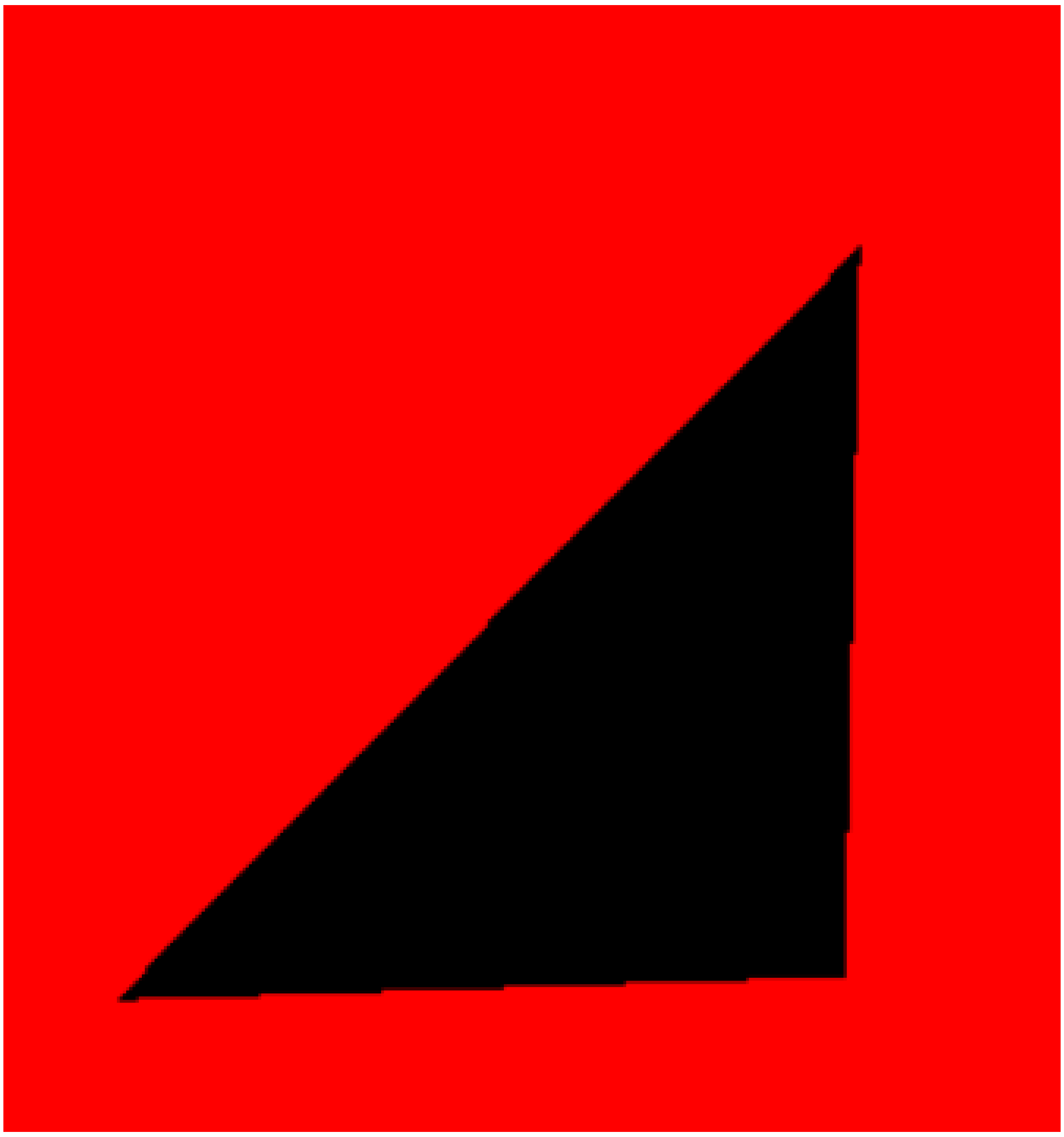


Figure 8: Result of 1.3 - 2D primitives.

1.4 - Addition of color

In this exercise we were asked to represent multiple objects as the ones in Figure 9.

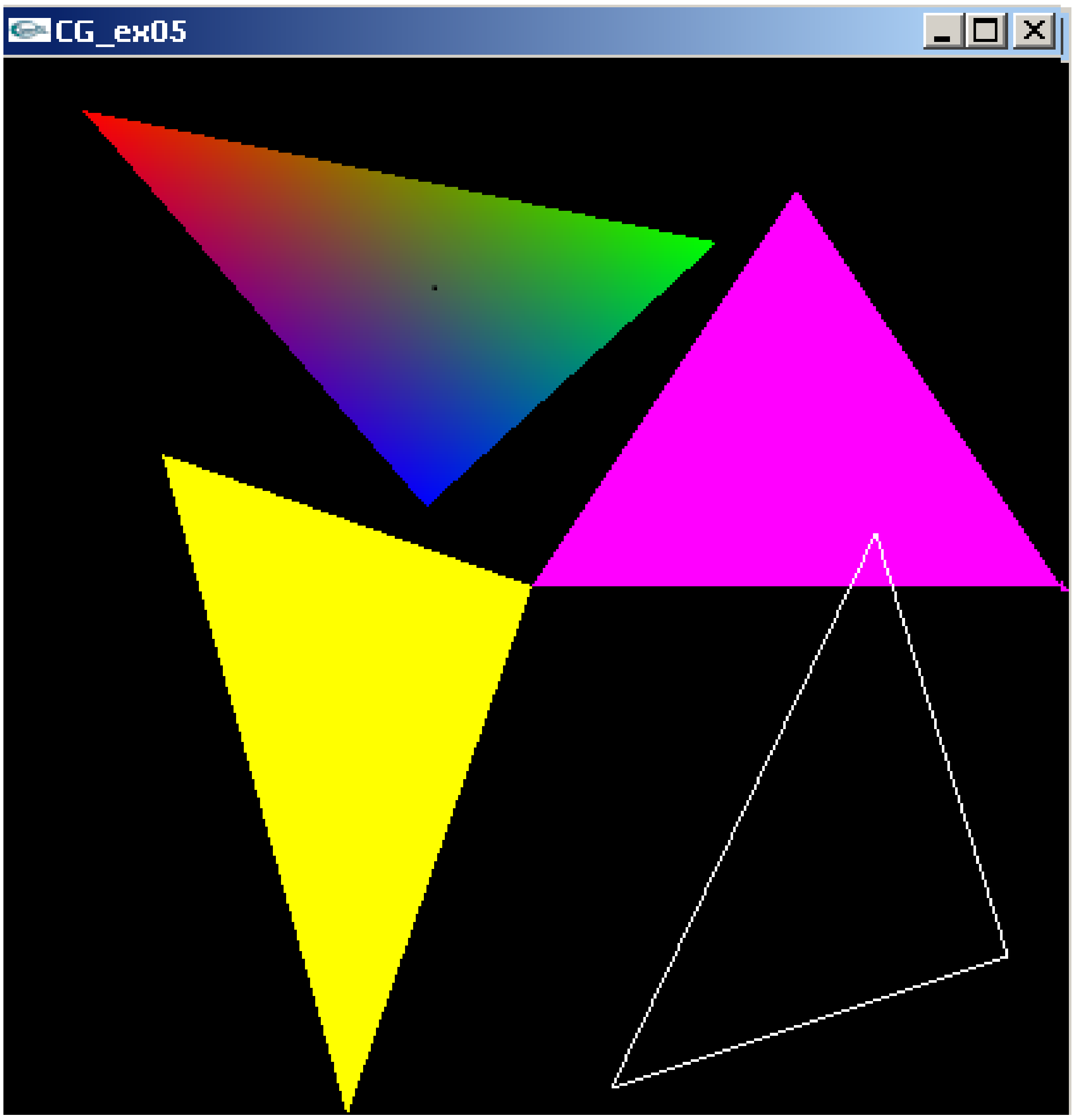


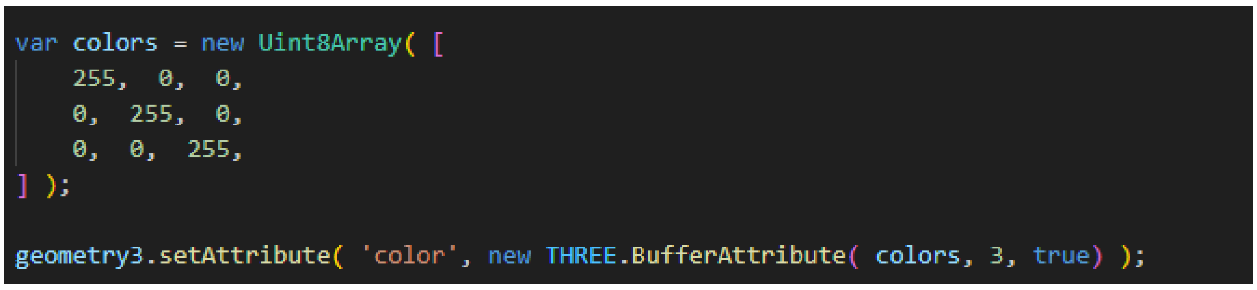
Figure 9: Example of 1.4 - Addition of color.

As it possible to notice, there are four triangles, three full painted and one with only the frame. In order to get this result, we decided to create 4 models, 1 for each triangle.

It was created a geometry buffer for each model as well as a color map for the triangle with multiple colors. It is necessary to associate a color to each vertex. The definition of the colors it is represented in Figure 10. It is important to mention that the vertices of each triangle follows the same syntax are the previews, like the one on section 0.4, and similar to the colors.

Some triangles were not seen because the face of the triangle with color was not the face that the camera was seeing. One way to fix this problem was using the "THREE.DoubleSide", which made both faces of the triangle use the color defined. The way we fixed it was by changing the order of the vertices, meaning that the triangle with be drawn in a different order and the face with color would be the face towards the camera.

For the frame triangle an extra parameter was implemented, the "wireframe:true". The code

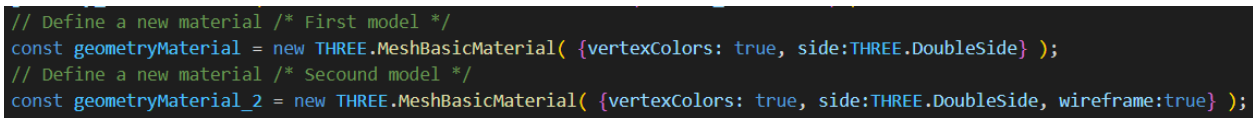


```
var colors = new Uint8Array( [
    255,  0,  0,
    0,  255,  0,
    0,  0,  255,
] );

geometry3.setAttribute( 'color', new THREE.BufferAttribute( colors, 3, true) );
```

Figure 10: Code for color definition of 1.4 - Addition of color.

is represented in Figure 11.

This time, the animation was removed.



```
// Define a new material /* First model */
const geometryMaterial = new THREE.MeshBasicMaterial( {vertexColors: true, side:THREE.DoubleSide} );
// Define a new material /* Secound model */
const geometryMaterial_2 = new THREE.MeshBasicMaterial( {vertexColors: true, side:THREE.DoubleSide, wireframe:true} );
```

Figure 11: Code for material definition of 1.4 - Addition of color.

The result obtained is represent on Figure 12. As it is possible to notice, our solution agrees with the example one, on Figure 9.

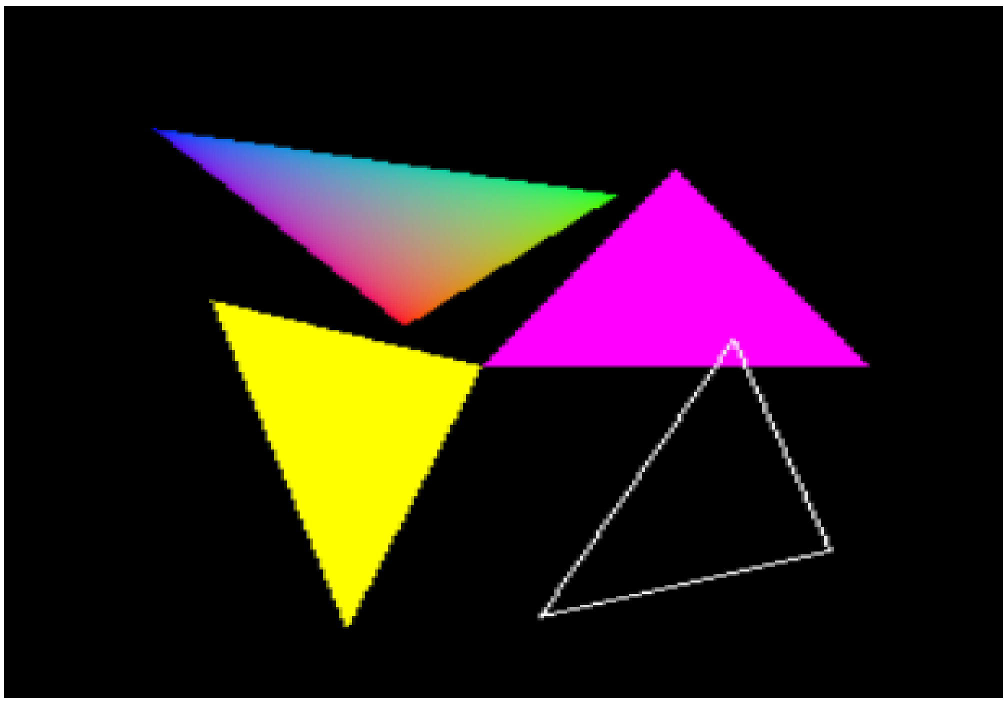
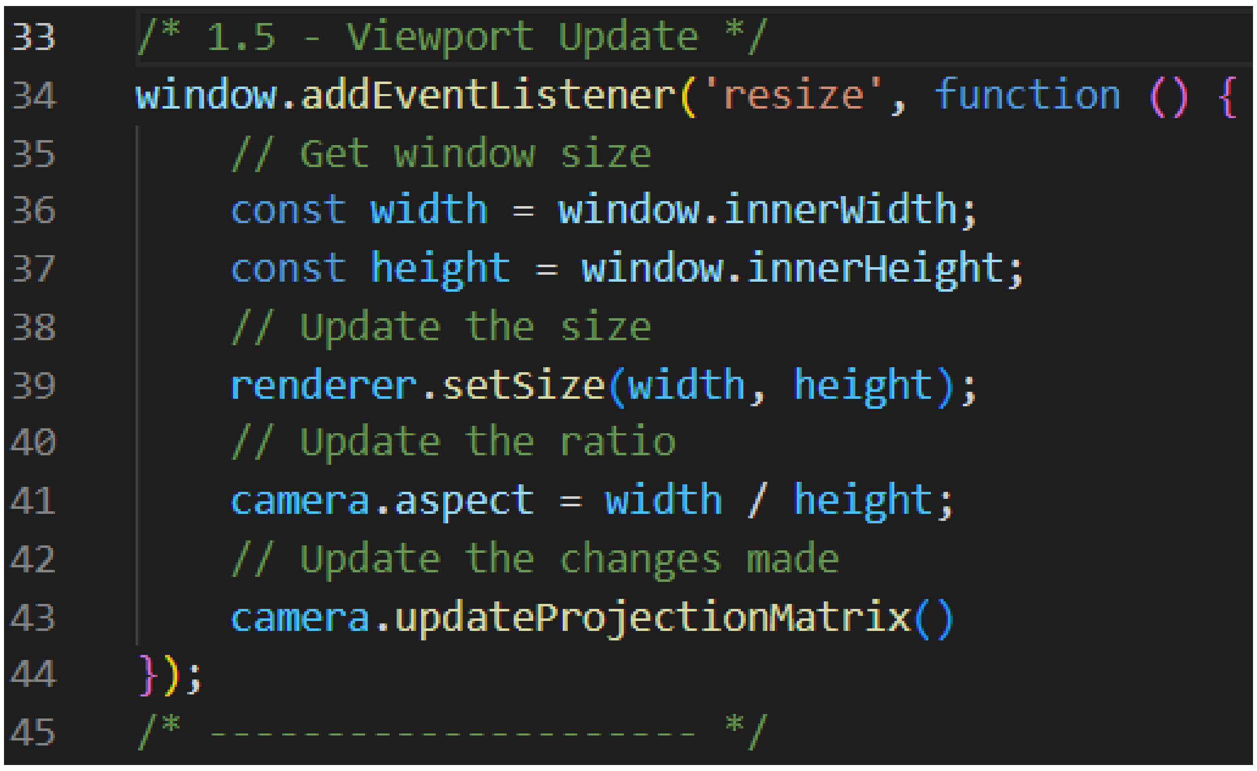


Figure 12: Result of 1.4 - Addition of color.

1.5 - View-port Update

For the penultimate we were challenged to change the dimensions of the browser window. We noticed that the visualization window is not update when the browser window size changes. In order to solve this problem, we were told to create a new function to be called when the browser window size is updated.

The function created as access to the the window size so it is possible to update size of the renderer accordingly. It also needs to modify the aspect camera ratio as well and update this change. The code to obtain this function is represented in Figure 13.



```
/* 1.5 - Viewport Update */
window.addEventListener('resize', function () {
    // Get window size
    const width = window.innerWidth;
    const height = window.innerHeight;
    // Update the size
    renderer.setSize(width, height);
    // Update the ratio
    camera.aspect = width / height;
    // Update the changes made
    camera.updateProjectionMatrix()
});
/* -------------------- */
```

Figure 13: Code for the function of 1.5 - View-port Update.

The result is illustrated on Figure 14 and, as it is possible to see, with the function developed on Figure 13, the left cube remains on the window. On the right, without the window update, the cube stays in the same position and if the window is re-sized. the cube despairs.

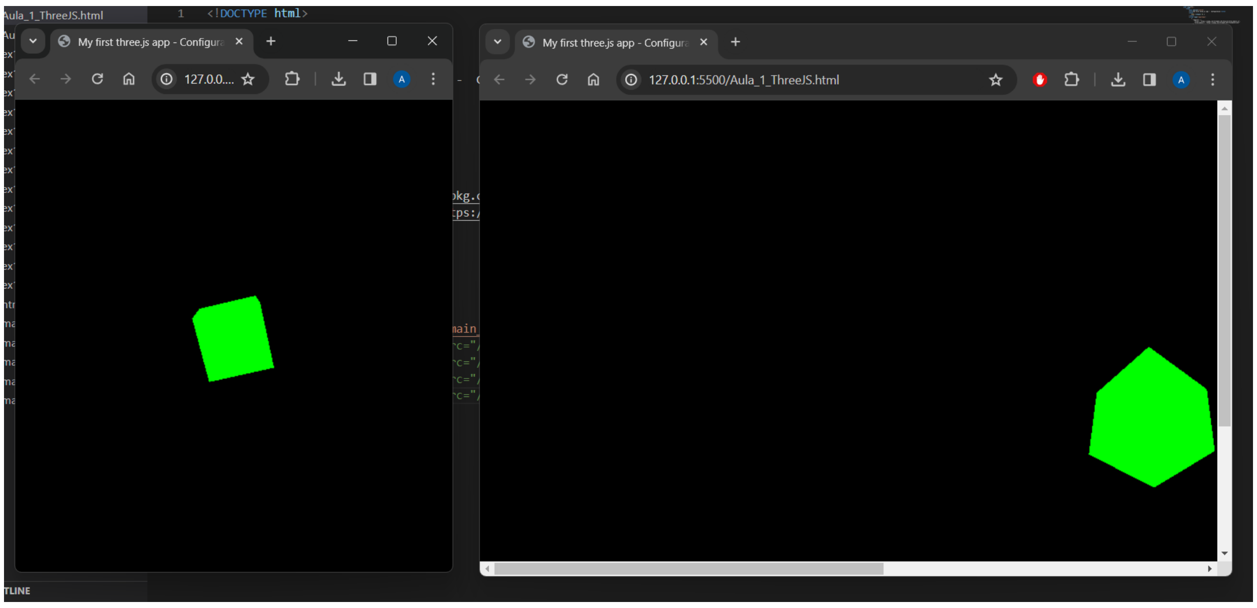

My first three.js app - Configur
127.0.0....
My first three.js app - Configur
127.0.0.1:5500/Aula_1_ThreeJS.html

Figure 14: Result of 1.5 - View-port Update.

1.6 - Other primitives

For the final exercise we were asked to modify the first example to show the cube in wireframe and investigate other available geometries and visualize at least 4 other geometries in the same scene changing some of their default parameters.

We have explored the geometries presented on Figure 15.

To achieve this geometries we follow the same logic as the Exercise 1.2 - First example. This time we've also explore a sphere, a cone, a cylinder, and a torus knot geometries.

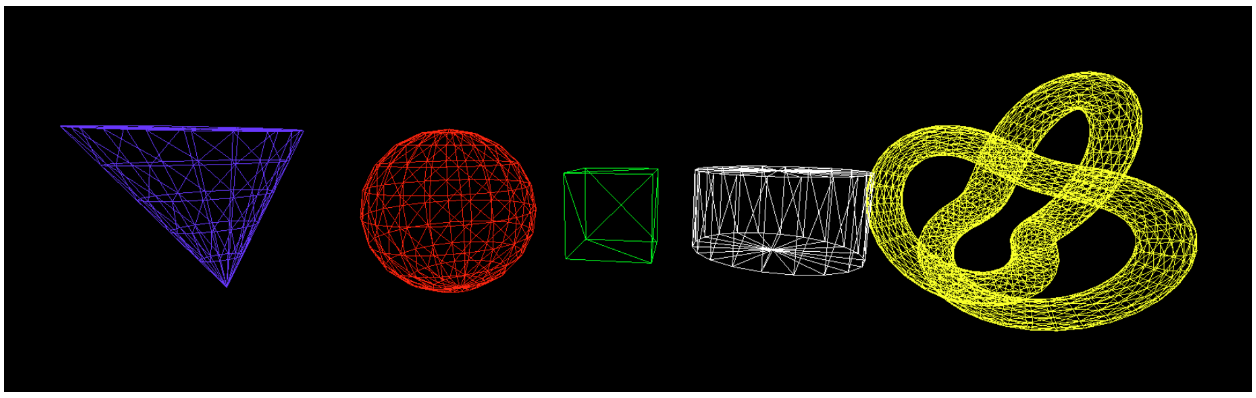
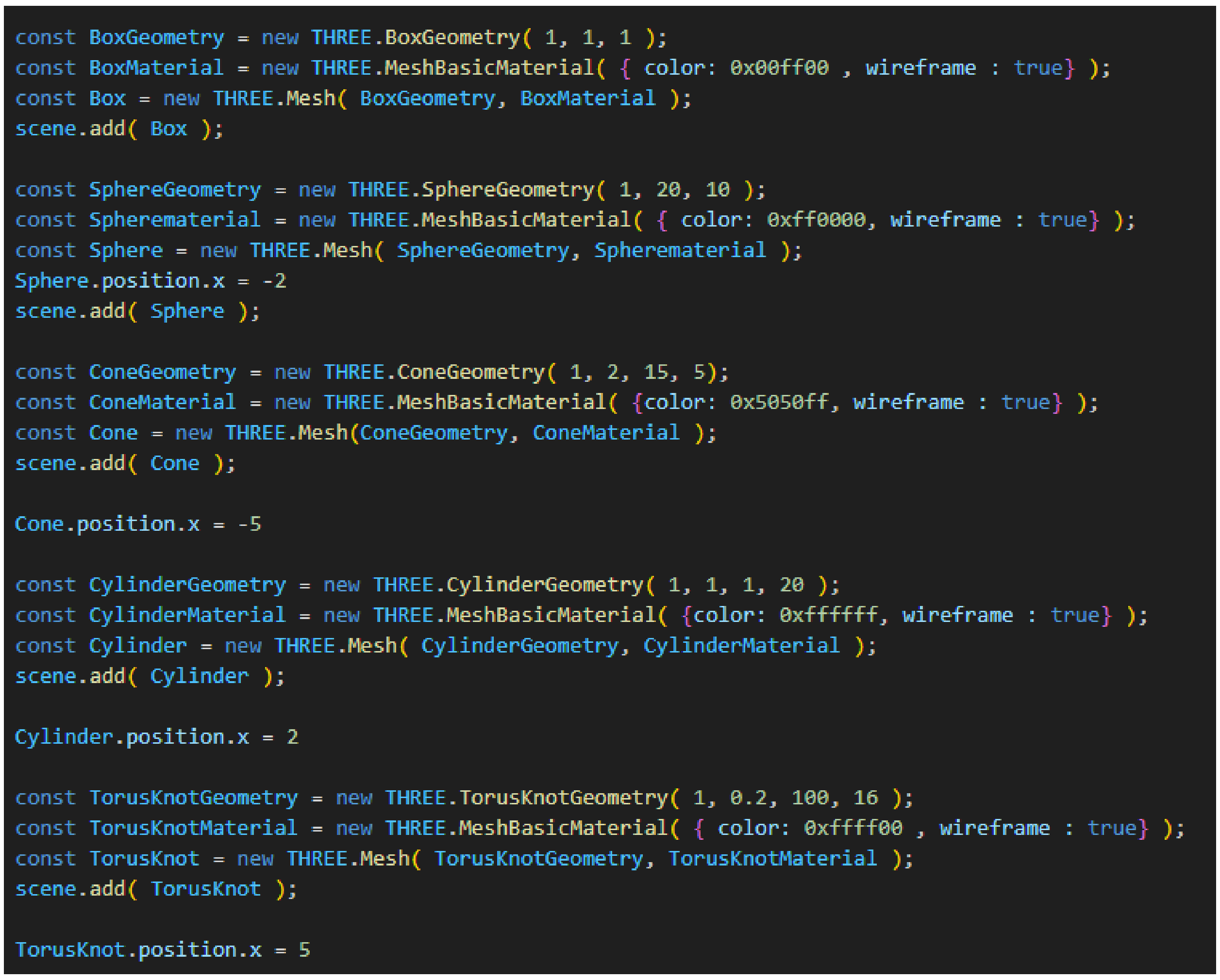


Figure 15: Result of 1.6 - Other primitives.

To get that shapes, we have used the code on Figure 16.



```
const BoxGeometry = new THREE.BoxGeometry( 1, 1, 1 );
const BoxMaterial = new THREE.MeshBasicMaterial( { color: 0x00ff00 , wireframe : true} );
const Box = new THREE.Mesh( BoxGeometry, BoxMaterial );
scene.add( Box );

const SphereGeometry = new THREE.SphereGeometry( 1, 20, 10 );
const Spherematerial = new THREE.MeshBasicMaterial( { color: 0xff0000, wireframe : true} );
const Sphere = new THREE.Mesh( SphereGeometry, Spherematerial );
Sphere.position.x = -2
scene.add( Sphere );

const ConeGeometry = new THREE.ConeGeometry( 1, 2, 15, 5);
const ConeMaterial = new THREE.MeshBasicMaterial( {color: 0x5050ff, wireframe : true} );
const Cone = new THREE.Mesh(ConeGeometry, ConeMaterial );
scene.add( Cone );

Cone.position.x = -5

const CylinderGeometry = new THREE.CylinderGeometry( 1, 1, 1, 20 );
const CylinderMaterial = new THREE.MeshBasicMaterial( {color: 0xffffff, wireframe : true} );
const Cylinder = new THREE.Mesh( CylinderGeometry, CylinderMaterial );
scene.add( Cylinder );

Cylinder.position.x = 2

const TorusKnotGeometry = new THREE.TorusKnotGeometry( 1, 0.2, 100, 16 );
const TorusKnotMaterial = new THREE.MeshBasicMaterial( { color: 0xffff00 , wireframe : true} );
const TorusKnot = new THREE.Mesh( TorusKnotGeometry, TorusKnotMaterial );
scene.add( TorusKnot );

TorusKnot.position.x = 5
```

Figure 16: Code for 1.6 - Other primitives.

As it possible to see, instead of "THREE.BoxGeometry", it was used "THREE.SphereGeometry", "THREE.ConeGeometry", "THREE.CylinderGeometry" and "THREE.TorusKnotGeometry". The animation is performed equal to each object.

Conclusion

To sum up, we consider that the Lesson 1 - three.js Introduction was concluded with success because all exercises were completed successfully and presented results were as expected.