Lesson 1 - three.js Introduction

Daniel Gomes
DETI, UA
Universidade de Aveiro
Aveiro, Portugal
Número Mecanográfico: 93015

Mário Silva
DETI, UA
Universidade de Aveiro
Aveiro, Portugal
Número Mecanográfico: 93430

***Abstract*—This report aims to describe the work developed in the context of the Information Visualization course of the University of Aveiro. Initially, an introduction to the *Three.JS* Technology is presented, being followed by the solution developed to a set of exercises proposed, in which the end result for each one of them is presented.**

***Index Terms*—Three.js, Primitives, Geometries, Scenes**

I. Introduction to Three.js

Three.js is a cross-browser JavaScript library and application programming interface (API) used to create and display animated 3D computer graphics in a web browser using WebGL. The source code is hosted in a repository on GitHub.

II. Environment Configuration

Three.js is a library built on WebGL to abstract some 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 following lines in the JavaScript code:

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

It is also possible to download *three.js* from http://threejs.org/ and use a local copy with the following link:

```
<script src="js/three.js"></script>
```

III. First Example

Following the example of Creating a Scene, we proceed to the development of a scene with a spinning cube, with the following steps:

- Scene Creation
- Camera Creation
- Renderer Creation
- Definition of an object/geometry and camera position
- Scene Render and Animation

In the context of *Three.JS*, a scene is what will manage the things that will be rendered to the user, such as objects, cameras, and others. In order to create and visualize a scene, we may write the following code snippet on a new JS script.

```
const scene = new THREE.Scene();
```

There are a few different cameras in three.js, but for this scene it was used a Perspective Camera, by using the following code:

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

The first attribute is the field of view. *FOV* 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. The next two attributes are the near and far clipping plane, objects further away from the camera than the value of far or closer than near won't be rendered.

The next step is the creation of the renderer, which allows rendering the content on the screen. In addition to creating the renderer instance, we also set the size at which we want it to render our example and the background color to white.

```
const renderer = new THREE.WebGLRenderer();
renderer.setSize(
    window.innerWidth, window.innerHeight);
renderer.setClearColor( "white", 1 );
document.body.appendChild(renderer.domElement);
```

Now, we are able to add objects items to our scene. In this case, we are creating a cube through the use of *BoxGeometry*, which contains all the vertices and faces needed. Recurring to the *MeshBasicalMaterial* allows us to customize the cube, such as its color. A Mesh is also necessary because it is an object that takes a geometry, and applies a material to it, which we then can insert to our scene, and move freely around.

By default, *scene.add()*, the thing we add will be added to the coordinates (0,0,0). This would cause both the camera and the cube to be inside each other. The camera has to be moved out a bit.

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

To render our scene and, consequently, the cube created before, we may use the following function. This will create a loop that causes the renderer to draw the scene every time the screen is refreshed. The use of *requestAnimationFrame* has some advantages, such as it pausing the rendering when the user changes to another browser tab.

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

After that, the render can be adapted to animate the cube. Here, we animated the cube by rotating him slowly on the x and y axis:

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

Finally, we get to visualize a spinning cube as we intended, as the following figure shows:

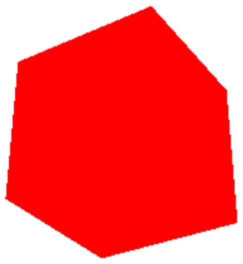

Fig. 1. Scene with a spinning cube

IV. 2D Primitives

The next exercise that was requested in this lesson, was the creation of a 2D black triangle using the coordinates (-1,-1,0) (1,-1,0) and (1,1,0), for each of the vertices. To do so, firstly, we may instantiate a new *BufferGeometry* object that will store the data needed to create the triangle, its coordinates, and reduces the cost of passing all this data directly to the GPU. To store the coordinates, it is used a *Float32Array* Data Structure. Next, we assign this data structure as the *position* attribute for the geometry object, using three values per vertices.

```
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, true));
geometryMaterial = new THREE.MeshBasicMaterial(
    {vertexColors: THREE.VertexColors});
var triangle = new
    THREE.Mesh(geometry, geometryMaterial);
scene.add(triangle);
```

The end product will be the triangle on the following figure.

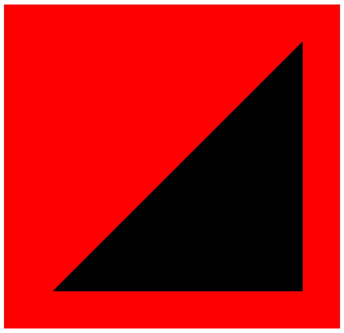

Fig. 2. Scene with a triangle

V. Addition of Color

To map different colors to the triangle, the following snippet of code can be used, where we create a *Uint8Array* data structure that will assign a color to each of the vertices of the triangle.

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

Creating multiple geometric objects in our scene is also possible. Similarly to the previous examples, a *Float32Array* should be used, where we store all sets of vertices that will be used for each of the objects. The last instruction on the snippet will use three values of the array per vertices, allowing, in the end, the creation of three different triangles.

```
var geometry = new THREE.BufferGeometry();
const vertices = new Float32Array([
    0.0, 0.0, 0.0,
    0.5, 0.75, 0.0,
    1.0, 0.0, 0.0,

    0.0, 0.0, 0.0,
    -0.35, -1.0, 0.0,
    -0.7, 0.25, 0.0,

    - 0.2, 0.15, 0.0,
    0.35, 0.65, 0.0,
    -0.85, 0.9, 0.0,
]);
geometry.setAttribute('position', new
    THREE.BufferAttribute(vertices, 3, true));
```

The fourth triangle that was suggested to create needed a different approach to be implemented since, due to its coordinates, it was not visible: only triangles with normal facing towards the camera are rendered. The solution was using another model, recurring the *wireframe* property, as it can be seen below.

```
geometry2.setAttribute('color', new
    THREE.BufferAttribute(color2, 3, true));
const geometryMaterial2 = new THREE.MeshBasicMaterial({
    vertexColors: THREE.VertexColors,
    side: THREE.DoubleSide,
    wireframe: true
});
var triangle2 = new
```

```
    THREE.Mesh(geometry2, geometryMaterial2);
scene.add(triangle2);
```

The end results can be seen on the following figure, where there are 4 different triangles.

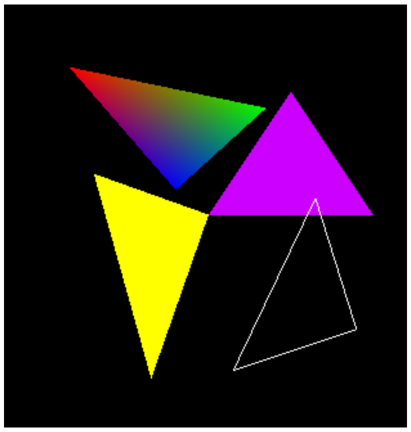

Fig. 3. Scene with several triangles

VI. Viewport Update

The following exercise has a goal solve a common problem when using *Three.JS*, that is the Browser windows resizing: when we change the dimensions of the Browser window, the objects that we create on a scene will be really difficult to be seen. Thus, the fix to the issue, can be use of an *Event Listener*, where every time the browser window size is changed, the camera and renderer sizes are updated. The following snippet contains this event listener.

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

VII. Other Primitives

Besides the Box and the Buffer Geometry, we also tested other primitives available from the *Three.JS* documentation. We tested a Sphere, Cone, Ring, and also a Torus Knot Geometry. We tested them around by changing some of their parameters, such as height, width, radius, color, number of segments, and, position. For example, on the sphere we set the radius 0.8, with 25 horizontal and vertical segments:

```
// sphere
geometry = new THREE.SphereGeometry(.8, 25, 25);
material = new THREE.MeshBasicMaterial(
 { color: "blue", wireframe: true }
);
sphere = new THREE.Mesh(geometry2, material2);
sphere.position.x = 2;
sphere.position.y = 0;
scene.add(sphere);
```

The end result, containing five different primitives, can be visualized on the figure bellow.

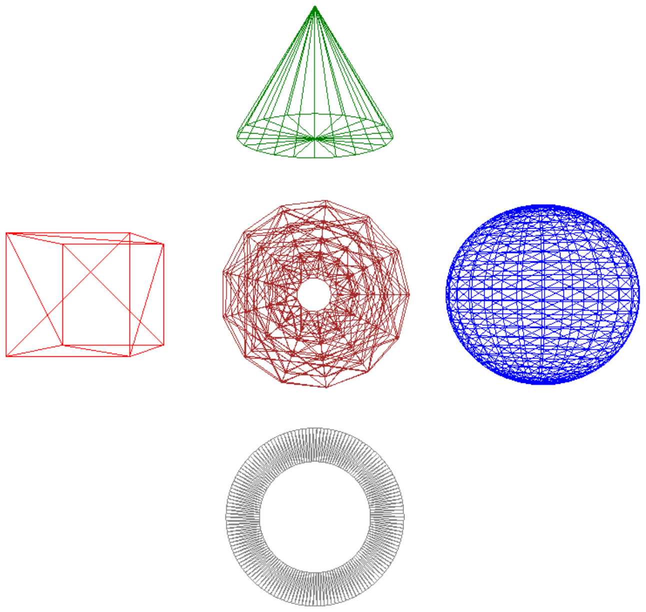

Fig. 4. Scene with Several Geometries

VIII. Conclusion

With this assignment, we managed to be introduced to the *Three.JS* technology and its basic functionalities. Besides this, we were able to understand its advantages such as the small amount of code needed to create visualizations, and the facility in writing it.