Three.js - Lesson 2

.

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 second 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

In computer graphics, lights and shades are very important tools that help the user better visualized the objects. They give depth to objects making them seem like a 3D object and not a 2D surface. The camera is also important, because, together with the other two, allows the user to see the world in another perspective. In this report we will show examples of how to implement lights and shades in the objects using Three.js.

II. Orthographic Camera

In the first example, we are asked to use an Orthographic camera and play around with its position using a camera control.

The first main difference from the last lesson, is that now when creating a camera, it is required to use a Orthographic Camera. With this type of camera, the object size is constant regardless of the distance from the camera. Can be very useful to understand the real measures of an object.

```
let viewSize = 10;
let aspectRatio = window.innerWidth /
    window.innerHeight;
let camera = new THREE.OrthographicCamera(
        -aspectRatio * viewSize / 2,
        aspectRatio * viewSize / 2,
        viewSize / 2,
        -viewSize / 2,
        0.1, 1000);
}
```

The camera is then created using the Three.js OrthographicCamera, which requires 6 frustum planes to describe the field of view of the camera. These planes are left, right, top, bottom, near and far. However, because the width and height of the screen are not usually the same, we need to calculate the field of view taking into consideration the aspect ration of the screen. It is also possible to define a view size in order to see the a bigger or smaller portion of the world.

This camera now has a problem when the window resizes which can be fixed by calculating the aspect ration after the resize.

```
window.addEventListener('resize', function () {
        renderer.setSize(window.innerWidth,
            window.innerHeight)
        let newAspect = window.innerWidth /
            window.innerHeight
        camera.aspect = newAspect;
        camera.left = viewSize * newAspect / -2;
        camera.right = viewSize * newAspect / 2;
        camera.updateProjectionMatrix();
});
```

The camera.aspect gives us the new aspect ratio and with that it is possible to change the x axis of the camera having into consideration the aspect ration.

The next task is to use a Camera Control which allows us to move the camera using certain controls. We used the OrbitControl for our example.

```
const controls = new THREE.OrbitControls(camera,
    renderer.domElement);
...
let render = function () {
        ...
        controls.update();

        renderer.render(scene, camera);
        };
```

The OrbitControl needs to be addded using a script tag. Then it can be created an instance using the OrbitControls constructor, which requires a camera and a renderer. In order for the controls to work, it is necessary to call the controls.update() method inside our render function.

III. Lighting

To test the effects of lighting in the cube example from last lesson, it was created a Directional Light. This type of light tries to simulate the light that we would get from the sun for example. The light rays are parallel and reach out to infinity.

```
const directionalLight = new
    THREE.DirectionalLight(0xffffff, 1.0);
directionalLight.position.set(0, 5, 0);
scene.add(directionalLight);
```

The previous snippet of code, creates a DirectionalLight object by providing the color and the intensity of the light. The light is then putted above the objects and its added to the scene.

There's another type of light added, the Ambient Light. This type of light is applied to every object in the scene and it's great for filling in areas on a render that do not have enough illumination.

```
const alight = new THREE.AmbientLight(0xffffff);
scene.add(alight);
```

The light is created passing its color and its added to scene. The results of adding the light are presented in Figure 1.

Fig. 1. Resultant cube affected with lighting.

IV. Shading

In this section, the objective is to create multiple spheres and observe the differences between shading on each of them.

A sphere in computer graphics is composed by multiple triangles. The more triangles the sphere has the closer its shape is to a real sphere. When calculating shading and lighting effects on a sphere, the number of triangles influence a lot the obtained result.

```
const sphereGeometry = new THREE.SphereGeometry(1,
    10, 10);
material = new THREE.MeshPhongMaterial({
        color: '#006063',
        specular: '#a9fcff',
        shininess: 100,
        flatShading: true
});
const sphere = new THREE.Mesh(sphereGeometry,
    material)
```

To create a sphere, we can use the SphereGeometry and provide its radius, the number of width segments and the number of heigth segments. The number of segments represent the number of triangles in the sphere, so the higher the values, the more realistic looking the sphere will be.

Then, it was added a directional light and a ambient light just like in the previous section.

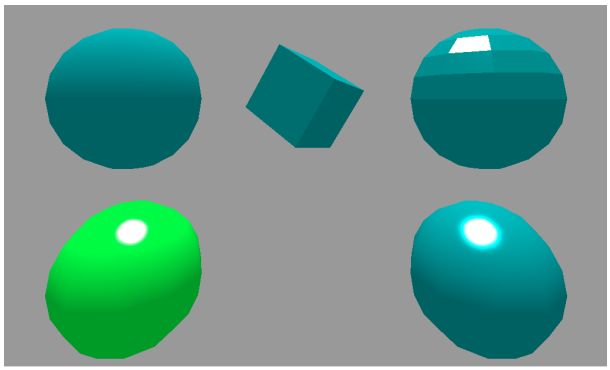

Fig. 2. Resultant spheres affected with shading.

In the right side, it was added 2 spheres with the same parameters but one of them had in the material, flatShading on and the other had it off. Flat shading used simple shading light techniques and so the object rendered does not look like a real sphere. Figure 2 has an example of how the use of flat shading can make the sphere look less natural, specifically the top right sphere.

It was tested the MeshLambertMaterial with the same characteristics of the other sphere but with specular and shininess components. Figure 2 shows the resulting sphere on the bottom right.

```
const emerald = new THREE.MeshPhongMaterial({
        shading: THREE.SmoothShading
});
emerald.color = new THREE.Color(0.07568, 0.61424,
    0.07568);
emerald.specular = new THREE.Color(0.633, 0.7278,
    0.633);
emerald.shininess = 0.6 * 256;
```

Another sphere was added, this time having an emerald material. Figure 2 shows the sphere on the bottom left.

Finally, multiple lights with different colors and positions were added and the final result looks like Figure 3.

V. Transparency

In order to test transparency, it was added a sphere wrapping each sphere of the last example. This outer sphere is however, transparent, which means that the sphere inside can be seen, although slightly different.

```
const glassMaterial = new THREE.MeshPhongMaterial({
        color: 0x222222,
        specular: 0xFFFFFF,
        shininess: 100,
```

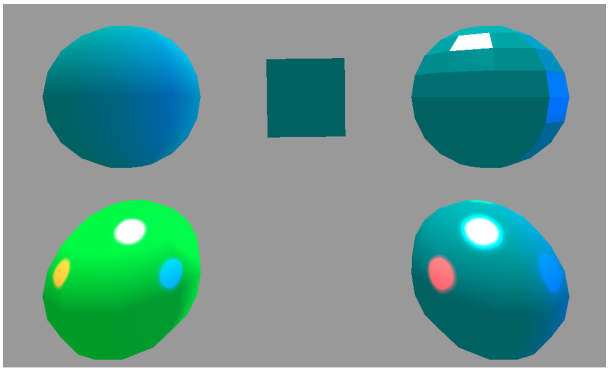

Fig. 3. Resultant spheres affected with red, green and blue directional lights.

```
        opacity: 0.5,
        transparent: true
});
```

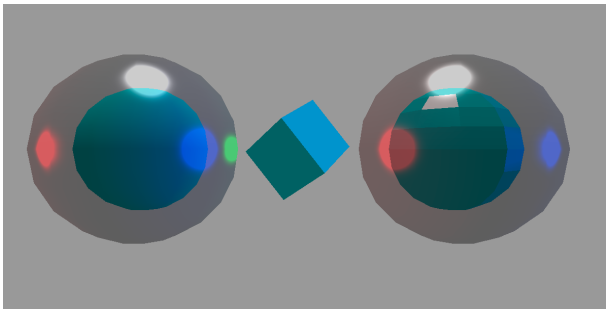

Fig. 4. Resultant previous spheres surrounded by transparency spheres.

The above material was used for those spheres. The result is shown in the Figure 4.

VI. Transformations

In another scene it was created a 3D object which can have multiple mesh associated with it. The transformations that are applied to the 3D object will be applied to each of its meshes.

```
const car = new THREE.Group();
```

First it was created the car object which will contain all the meshes.

```
const boxGeometry = new THREE.BoxGeometry(2, 1, 4);
let material = new THREE.MeshPhongMaterial({
    color: 0x0066ff,
    specular: '#a9fcff',
    shininess: 0,
    opacity: 0.5,
    transparent: true,
});
const cube = new THREE.Mesh(boxGeometry, material);
car.add(cube);
```

Then we started by adding the car body using a box geometry.

```
const wheelGeometry = new
    THREE.CylinderGeometry(0.5, 0.5, 0.2, 32);
const glassMaterial = new THREE.MeshPhongMaterial({
    color: 0xff0000,
    specular: 0xFFFFFF,
    shininess: 0,
    opacity: 0.5,
    transparent: true
});
for (const pos of [[-1, -0.5, -2], [1, -0.5, 2],
    [-1, -0.5, 2], [1, -0.5, -2]]) {
    let sphere = new THREE.Mesh(wheelGeometry,
        glassMaterial);
    sphere.position.set(...pos);
    sphere.rotation.z = 0.5*Math.PI;
    car.add(sphere);
}
```

After the body, each of the wheels was added to its corresponding position. The car finally has all its parts (Figure 5.

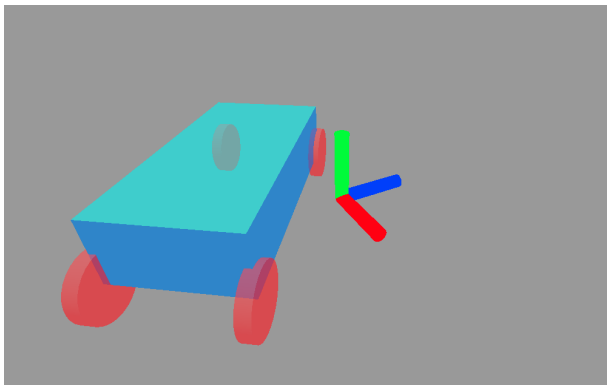

Fig. 5. Resultant car and axes as cylinders.

In order to make the car move around a pivot point, we first moved the car to this pivot point, subtracted the pivot point from the car's original position, and then applied the rotation. Then, we moved the car back by the same position vector and added the pivot point to the car's new position. All this on the render cycle.

VII. 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.