Lesson 2 - Projections, lighting and transformations

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.**

***Index Terms*—Three.js, Camera Models, Lighting, Shading, Transformations**

I. Camera models

For the first exercise, we used a simple scene with a cube, but instead of using a perspective camera, we used an Orthographic Camera.

In this projection mode, an object's size in the rendered image stays constant regardless of its distance from the camera.

Orthographic projection is a method of showing three-dimensional objects in two dimensions (that is, on a plain surface). It is a form of parallel projection, in which all the projection lines are orthogonal to the projection plane, resulting in every plane of the scene appearing in affine transformation on the viewing surface.

In this type of camera, the upper limits are specified in x and y coordinates on the scene and not with the field of view angle as in the perspective camera. We modified the parameters so that the world view is between -3 and 3 on the x-axis while respecting the window's aspect ratio.

```
var camera = new THREE.OrthographicCamera(
  -3 * aspect, 3 * aspect, 3, -3,  0.1, 1000
);
```

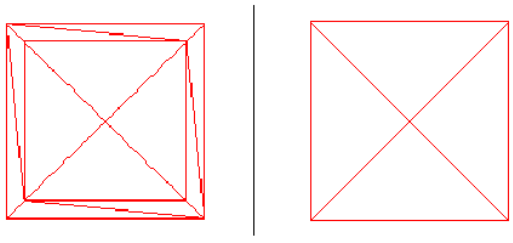

Fig. 1. Two Scenes with a Perspective (left) and an Orthographic (right) Cameras

From the figure above 1, we can visualize that the orthographic camera, on the right, did project the cube in a plain surface as expected, and the perspective, left, allows having the notion of distance, as the figure 4 below demonstrates.

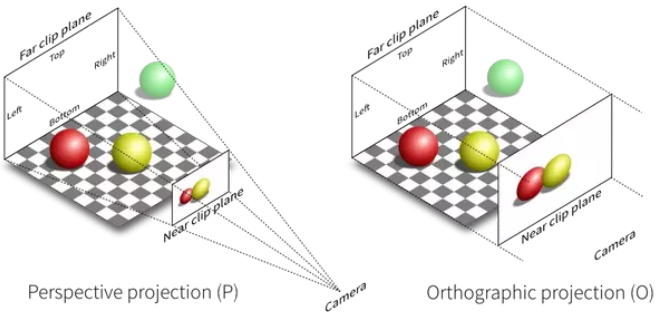

Fig. 2. Two Scenes with a Perspective (left) and an Orthographic (right) Cameras

We also added a function to ensure that the cube aspect ratio does not change when the window is resized:

```
window.addEventListener('resize', function() {
  var w = window.innerWidth;
  var h = window.innerHeight;
  var aspect = w / h;
  camera.left   = - 3 * aspect;
  camera.right  = 3 * aspect;
  camera.top    = 3;
  camera.bottom = -3;
  camera.updateProjectionMatrix();
  renderer.setSize(w, h);
})
```

II. Orbit control

Three.js provides classes to allow an easy control of the camera pose. We tested the Orbit controls, that allows the camera to orbit around a target, but while keeping a particular axis locked. This is extremely popular because it prevents the scene from getting "tilted" off-axis.

```
var controls = new THREE.OrbitControls(
  camera, renderer.domElement
);
```

And then update the camera controls in the render function:

```
controls.update();
```

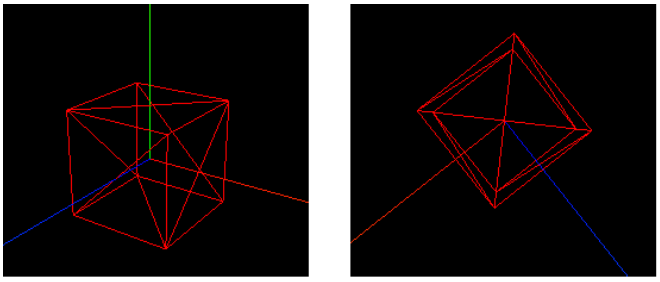

Fig. 3. Scene with Orbit Control

As it can be seen on the figure above **??**, some axis were added in order to help visualization, and then we tried to make the green helper axis facing down, but it was not possible using the Orbit Control.

III. Trackball Control

We also tried other camera control, the Trackball Control, which is similar to Orbit Controls. However, it does not maintain a constant camera up vector. That means if the camera orbits over the "north" and "south" poles, it does not flip to stay "right side up".

On the figure below 10, it demonstrates the green helper axis facing down, that wasn't possible before, but with this camera we were able to rotate in every direction.

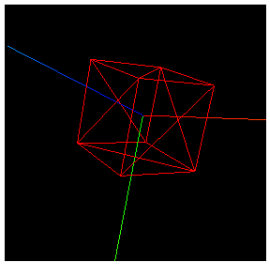

Fig. 4. Scene with Trackball Control

IV. Lighting and materials

There are three kinds of light:

- ambient;
- diffuse;
- specular.

Ambient light is constant from all directions, diffuse depends on the angle between the light and the surface and specular depends on the angle between the reflected incoming light and the eye. It also depends on a scalar quantity called shininess.

In Three.js, the color attribute of a material means how it interacts with diffuse light. The ambient and specular properties are named in the obvious way. In Three.js, we can have:

- Point lights;
- Ambient lights;
- Directional lights;
- Spotlights;
- Others...

So for this exercised, it was created a Directional Light at position 0.5.0 with color 0xffffff and intensity 1.0.

A directional light casts parallel light rays throughout the scene. It can be thought of as a point source infinitely far away in a particular direction, and indeed, the mathematical model of it is like that.

In order for the object to interact with light, it is necessary to use a material of a different type from MeshBasicMaterial, so we used a MeshPhongMaterial material.

Ambient light is the light all around us. In most real-world scenes, there is lots of light around, bouncing off objects and so forth. We can call this ambient light: light that comes from nowhere in particular. It globally illuminates all objects in the scene equally, and it cannot be used to cast shadows as it does not have a direction. We added it with the following code:

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

Resulting in the following scene of the figure 5:

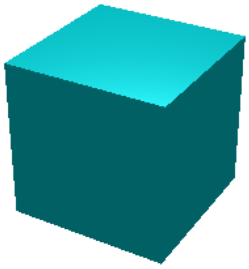

Fig. 5. Scene with a directional and an ambient light

A spotlight is just a point source limited by a cone of angle θ. Intensity is greatest in the direction that the spotlight is pointing and trailing off towards the sides, but dropping to zero when it is read the cutoff angle θ. This light gets emitted from a single point in one direction, along a cone that increases in size the further from the light it gets. This light can cast shadows.

V. Shading

For the shading exercise, it was used the geometry of a shepere, *sphereGeometry*. The parameter *widthSegments* is the number of horizontal segments and the *heightSegments*, the number of vertical segments. We used 10 segments for this exercise.

We placed two spheres side by side and added the ambient light and directional light from the previous exercise located between the two spheres, and applied the same *MeshPhong-Material* to the spheres. We tested the *flatShading* option of one of the materials by toggling between true and false, as the figure 6 demonstrates.

Fig. 6. Scene with two different flat shading options

The sphere on the left has flat shading on false, and the right one has the flat shading option activated. What it does is basically allow shading on the flat surfaces of the sphere, and therefore it makes the edges of the segments more perceivable, on the other hand, when it's not activated, it makes the sphere appear more smooth.

As an extra, demonstrated on figure below 7, we applied to the left sphere a *MeshLambertMaterial* type material. This material scatters light evenly in all directions, so the specular coefficient and brightness are ignored. We also turned off the flat shading on the right sphere, and added a red directional light, a blue directional light and a green spotlight light in different positions.

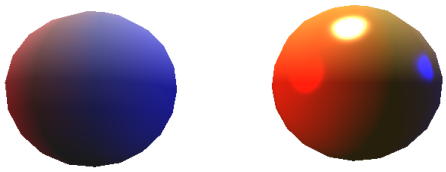

Fig. 7. Extra scene with more lights

VI. Transparency

The next exercise addressed an attribute that can be added to our objects, transparency. To do so, regarding the spheres that have been created on the previous exercise, we have added two additional transparent spheres in similar positions but with a slightly greater size. The snippet bellow shows the material in these two spheres.

```
const glassMaterial = new THREE.MeshPhongMaterial({
    color: 0x222222,
    specular: 0xFFFFFF,
    shininess: 100,
    opacity: 0.25,
    transparent: true
});
```

As an end result we managed to obtain the following

Fig. 8. Scene with Transparency

VII. Tranformations (scale and rotation)

Next up, we managed to create a paralleliped with 4 spheres centered on its lower vertices, recurring to the scale and rotation properties. In the end, the final result might look similar to a really abstract car. To do so, instead of adding multiple separated meshes for each of the figures to the scene, we added these meshes to a single *THREE.Object3D()*, denominated *car*.

```
const car = new THREE.Object3D()
```

The creation of the paralelliped was similar to previous exercises where we used a sphere, with the difference that the box size was set through the scale property, as it can be seen bellow. This property contains an instance of vector3 that by default will contain a value of one for each axis.

```
const geometry = new THREE.BoxGeometry();
const glassMaterial = new THREE.MeshPhongMaterial({
    color: 0x00aaff,
    specular: 0xFFFFFF,
    shininess: 60,
    opacity: 0.5,
    transparent: true,
});
const parallelepiped = new THREE.Mesh(geometry,
glassMaterial);
parallelepiped.scale.set(2, 1, 4)
car.add(parallelepiped)
```

The following step consisted in the creation of the spheres, which we may identify as the "wheels" of a car. Thus, we create 4 different meshes that are added to the *car* Object3D.

```
scene.add( car );
```

Finally, to render all objects, we add the **car** object to our scene. As an end result, we obtained the following representation:

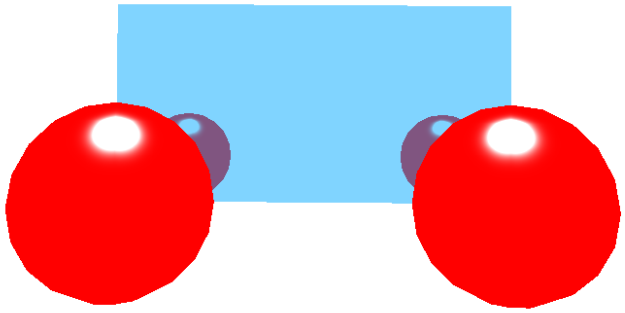

Fig. 9. Scene with Transformation

VIII. Tranformations (rotations)

The remaining exercise consisted on creating an object that represented the Coordinate System through the use of cylinders for each of the axis x,y and z. Next up this object should be added to the previously created scene and finally animate the movement of the ”car”. Firstly, to create the Coordinate System representation object, three cylinders have been created through the use of a *CilinderGeometry Object*. In the y and z axis Cylinders, it was necessary to recur to rotations in order to put these in the correct direction.

```
const axis_material_y=new THREE.MeshPhongMaterial({
    color: '#ff0000',
    specular: '#a9fcff',
    shininess: 100,
    flatShading: false
});
const axis_y = new THREE.Mesh(
    axis_cylinder,
    axis_material_y
);
axis_y.rotation.z = 1.6
axis_y.position.y = -0.5
axis_y.position.z = -0.5
axis.add(axis_y)
const axis_material_z=new THREE.MeshPhongMaterial({
    color: '#0000ff',
    specular: '#a9fcff',
    shininess: 100,
    flatShading: false
});
const axis_z = new THREE.Mesh(
    axis_cylinder,
    axis_material_z
);
axis_z.rotation.x = 1.6
axis_z.position.y = -0.5
axis_z.position.x = -0.5
axis.add(axis_z)
scene.add(axis)
```

Next, to make the spheres more similar to a wheel shape, the Geometry object (to be used by all wheels) has also been replaced to a Cylinder Geometry.

```
const wheels = new THREE.Object3D()
const wheel = new THREE.CylinderGeometry(
    0.5, 0.5, 0.2, 14
);
const wheel_material = new THREE.MeshPhongMaterial({
    color: '#CC0000',
    specular: '#a9fcff',
    shininess: 40, flatShading: false,
    opacity: 0.55,
    transparent: true
});
```

Each of the wheels, have also used the ”rotation” property so that each one of them was positioned on the right direction in all lower vertices of the parallelized. To make an animation in the “car” object, we have decreased the z axis value in the *animate()* function. Therefore, every time the objects are rendered, its “z” value is updated.

```
const animate = function () {
requestAnimationFrame(animate);

// animate the car
car.position.z -= 0.01
axis.position.z -=0.01

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

The end product can be visualized on the figure bellow.

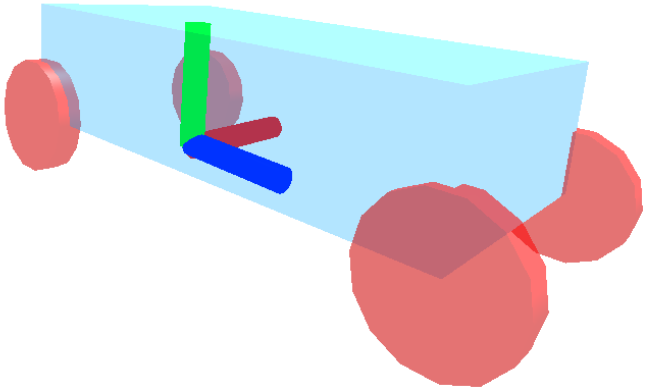

Fig. 10. Scene with Transformation

IX. Conclusion

With this assignment we managed to gather more knowledge on the *Three.js* Technologies, and more how to do projections, manage the lighting on our scenes and create transformations. Without doubt that some of these exercise have been more challenging, which is important to develop our skills while using *Three.js*.