ThreeJS, Lesson 2 - Projections, lighting and transformations

Ivo Félix, 109641, MSc Software Engineering
Rita Ferrolho, 88822, MSc Computational Engineering
Information Visualization, 2019 University of Aveiro

Camera models

The code written for this section is in Camera models.htm.
In order to visualize the cube from the first example from the last lesson in wireframe, it is necessary to state that wireframe:true, in the material object, as shown in the following line of code:

```
const material = new
THREE.MeshBasicMaterial({ color:
0x00ff00, wireframe: true });
```

To disable the cube rotation, the following lines were commented inside the render() function:

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

The cube, which is being projected with a Perspective Camera, is shown in the following figure:

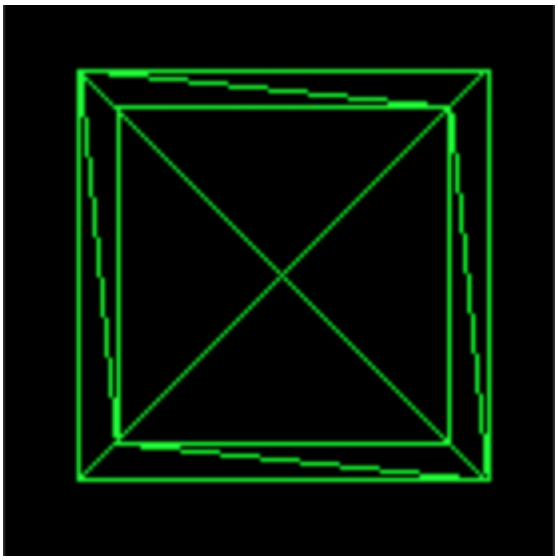

Figure 1 – Wireframe cube, projected with a Perspective Camera.

The camera object created with the Perspective Camera is shown in the following line of code:

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

The first parameter of the constructor, with value 75, corresponds to the camera's vertical field of view (the bigger the value, the smaller the cube gets, and vice versa). The second

parameter corresponds to the camera aspect ratio. Typically, the ideal value is the ratio of the width of the browser window to its height. As for the last two parameters, they represent the minimum and maximum distance of the object to the camera at which the object is shown, respectively.

If the cube is projected with an Orthographic Camera, the following result can be obtained:

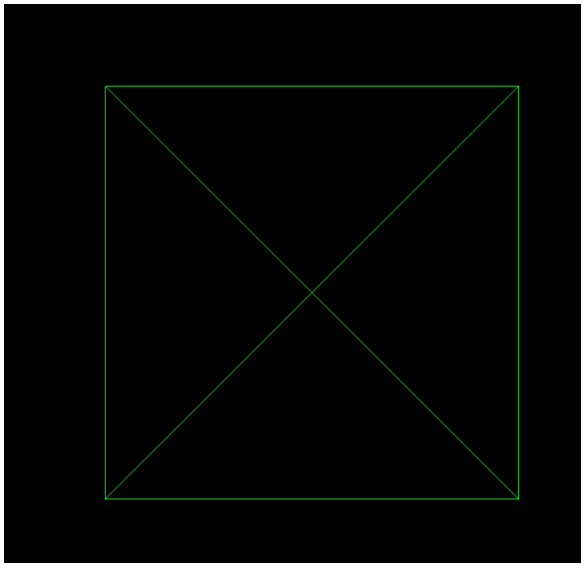

Figure 2 – Wireframe cube, projected with an Orthographic Camera.

A difference can be spotted in the way a perspective camera projects a cube, in comparison to the projection of the same object by an orthographic camera. In the perspective camera, all the wires of the cube, including vertices and edges, are shown. With an orthographic camera, however, only the front face of the cube is shown. The reason for this is that, while in a perspective camera, parallel lines are closer the farther away they are to the observer (at infinity, they seem to merge to a dot), in an orthographic camera, parallel lines are always represented as such, independently of their distance to the observer. A perspective camera is recommended for realistic representation of objects, as they mimic the human eye system. An orthographic camera, however, is recommended for knowing the exact dimensions of an object. The figure below demonstrates both projections.

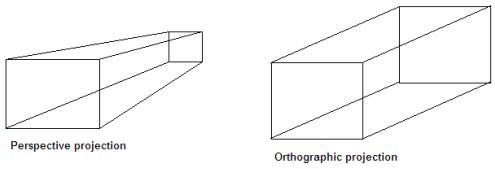

Figure 3 – Perspective and Orthographic projections.

The camera created to reproduce the cube with an orthographic projection is written as the following:

```
const camera = new
THREE.OrthographicCamera(
window.innerWidth / - 1000,
window.innerWidth / 1000,
window.innerHeight / 1000,
window.innerHeight / - 1000);
```

As one might notice, the parameters of this constructor are different from the Perspective projection. These parameters represent the left, right, top and bottom plane of the camera, respectively.

In order to update the viewport after resizing the browser window, the following code was added:

```
window.addEventListener('resize'
, onWindowResize); // event
listener that resizes viewport
whenever the browser window
changes

function onWindowResize() {

camera.aspect =
window.innerWidth /
window.innerHeight; // update
camera aspect ratio

camera.updateProjectionMatrix();
// update camera projection


renderer.setSize(window.innerWid
th, window.innerHeight); //
update renderer dimensions

}
```

Orbit control

The code written for this section is in Orbit control.htm.
To add orbit control, the following code was added:

```
const controls = new
THREE.OrbitControls(camera,
renderer.domElement); // outside
the update() function
// …
controls.update(); // inside the
update() function
```

When the user clicks and drags on the mouse, it gives an ilusion that they are rotating the object. However, it is the camera that is moving around the object, “looking” at it from different angles. The logs in the console browser show the vectors which represent the direction at which the camera is looking at as a vector, this illustrates that the objects themselves remain stationary and only the camera is being moved.

Lighting and materials

The code written for this section is in Lightning and materials.htm.
If we go back to the first example from the last lesson, and add a directional light, there are no changes in the scene. To add the directional light, the following code was written:

```
       const directionalLight =
new
THREE.DirectionalLight(0xffffff,
1.0);

scene.add(directionalLight);
```

There are no changes in the scene because it is necessary to use a material of different type. In order for the object to interact with light, the material of type MeshBasicMaterial should be replaced by MeshPhongMaterial, for example. The new material is created in the following code:

```
const material = new
THREE.MeshPhongMaterial({
```

```
color: '#006063',

specular: '#a9fcff',

shininess: 100

});
```

Without any lightning, an object with a material of type MeshPhongMaterial is not visible. With directional lightning, however, some parts of the object can be seen. In the case of the cube, two faces of it can be seen, at most, as shown in the figure below.

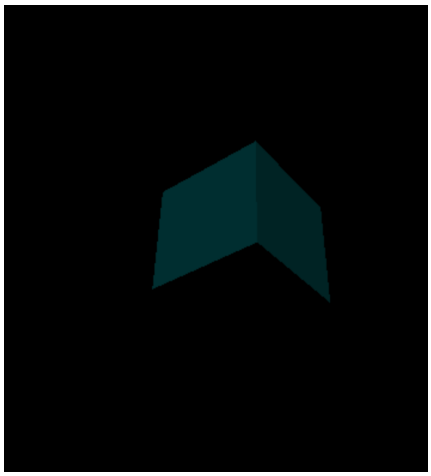

Figure 4 – Cube with MeshPhongMaterial and directional light.

An ambient light was then added, by typing the following code:

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

scene.add(alight);
```

After adding the ambient light, the three front faces of the cube can be seen. The cube has gotten brighter, as well:

Figure 5 – Cube with MeshPhongMaterial, directional light and ambient light.

The ambient light is the diffuse component of the light, that is, the light component that does not depend on direction. The property ‘color’ defined for a material in Three.js only interacts with diffuse light, which is why all faces of the cube are not visible, instead of just the ones illuminated by the directional light.

Shading

To modify the rotating cube example to a sphere, the following code was used, written in Shading 1 - wireframe sphere.htm, where the first parameter corresponds to the radius of the sphere:

```
const geometry = new
THREE.SphereGeometry(1, 10, 10);
```

With the wireframe option active, if the second parameter is reduced to one, the obtained result is as presented in the figure below. This parameter corresponds to the widthSegments.

Technically, the number of width segments being used is actually 3, since it is impossible to obtain a shape with less than 3 width segments. So, if the second parameter is specified by an integer lower than 3, the number of width segments used is 3.

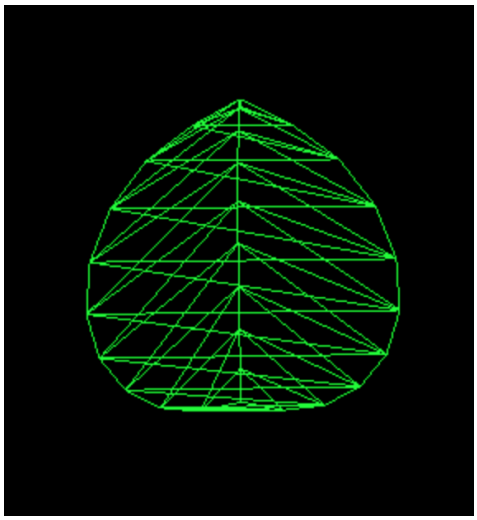

Figure 6 – A sphere with radius 1, witdthSegments 1 (technically, there are three width segments) and heightSegments 10.

If the second parameter (widthSegments) is changed back to its previous value, which is ten, and if the third parameter is reduced to one, the obtained result is as shown in the figure below. This parameter corresponds to the heightSegments.

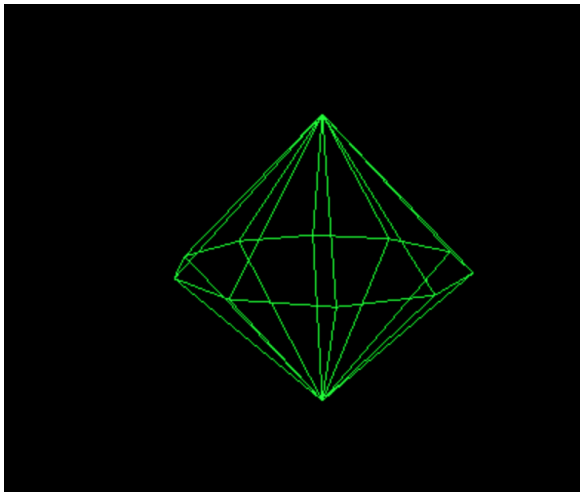

Figure 7 – A sphere with radius 1, witdthSegments 10 and heightSegments 1.

After disabling the wireframe and setting the number of segments to 10 (Shading 2 - two spheres.htm), another sphere was created, with similar characteristics. To do so, the same geometry and material was applied to the new sphere. Then, the position of the spheres was changed so that one would be at x=2.5 and the other at x=-2.5. The developed code to accomplish this is typed down below:

```
        const sphere1 = new
THREE.Mesh(geometry, material1);
        sphere1.position.x =
-2.5;
        const sphere2 = new
THREE.Mesh(geometry, material1);
        sphere2.position.x =
2.5;
        scene.add(sphere1);
        scene.add(sphere2);
```

The obtained scene looks like the following:

Figure 8 – Two spheres with similar geometry and material (MeshBasicMaterial).

By adding an ambient lightning, directional lightning located at y=5, and changing the material of the spheres to MeshPhongMaterial (if the previous material were to be used, the lightning added would be unnoticed. Also, this new material can only be seen when lightning is added), the following scene is obtained:

Figure 9 – Two spheres with similar geometry and material (MeshPhongMaterial), with additional ambient and directional lightning.

By adding the flatShading option to the material of the right sphere, the scene is updated to the following:

Figure 10 – Two spheres with similar geometry and material (MeshPhongMaterial), with additional ambient and directional lighting. The sphere on the right has the flatShading option activated.

By default, when materials in Three.js interact with light, the results for each point on one face of a surface come from the interpolation of the normal of the surface to give a smoother and more realistic appearance. By disabling this interpolation (by enabling the flatShading option) the illumination of the entire surface is the same, which is why on the sphere on the right each face is much more distinct than on the sphere on the left.
If the MeshLambertMaterial type material is applied to the left sphere (Shading 3 - lambertian material and other phong parameters.htm), with the same material properties, the following scene is obtained:

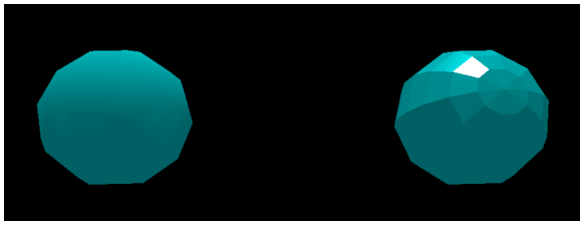

Figure 11 - Two spheres with similar geometry, with additional ambient and directional lighting and with flatShading option set to true on the right sphere. The left sphere uses MeshLambertMaterial while the right sphere uses MeshPhongMaterial.

If the specular and shininess components are removed of the Lambertian-type material, no difference

can be spotted in the scene. The reason for that is because Lambertian materials scatter light evenly in all directions, as opposed to the MeshPhongMaterial.

By modifying the materials properties so that one sphere seems to be made of emerald and the other of gold, the following scene was obtained:

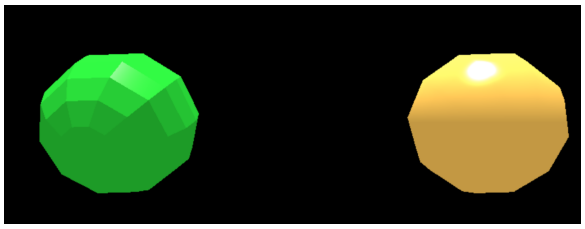

Figure 12 - Two spheres with similar geometry, with additional ambient and directional lighting. Both use the same material type, but the left sphere mimics the properties of emerald and has flatShading activated. The right sphere material resembles gold.

In the next step, another three directional lights were added: a red light, a blue light and a green light. The code that defines and adds these lights is shown below:

```
	dirLight2 = new THREE.DirectionalLight(0xff0000, 1.0); // red light
	dirLight2.position = new THREE.Vector3(-5, 0, 0); // red light position
	scene.add(dirLight2); // add red light to the scene
	dirLight3 = new THREE.DirectionalLight(0x0000ff, 1.0); // blue light
	dirLight3.position = new THREE.Vector3(5, 0, 0); // blue light position
	scene.add(dirLight3); // add blue light to the scene
	dirLight4 = new THREE.DirectionalLight(0x00ff00, 1.0); // green light
	dirLight4.position = new THREE.Vector3(0, 0, -5); // green light position
	scene.add(dirLight4); // add green light to the scene
```

Since all directional lights point to the origin by default, there is no need to set target in the lights.

Transparency

The code written for this section is in Transparency.htm.

In order to add two transparent cubes surrounding the spheres, the first step was to define the following cube geometry, where all edges have size 3:

```
	const cubeGeom = new THREE.BoxGeometry(3, 3, 3);
```

Then, the glass material was defined:

```
	const glassMaterial = new THREE.MeshPhongMaterial({
		color: 0x222222,
		specular: 0xFFFFFF,
		shininess: 100,
		opacity: 0.3,
		transparent: true
	});
```

Then, two cubes were created and added to the scene, with the same geometry and material, but in different

positions (the first cube is positioned at x=-2 and the second at x=2):

```
	const cube1 = new
THREE.Mesh(cubeGeom,
glassMaterial);
	cube1.position.x = -2;
	scene.add(cube1);
	const cube2 = new
THREE.Mesh(cubeGeom,
glassMaterial);
	cube2.position.x = 2;
	scene.add(cube2);
```

The obtained scene was the following:

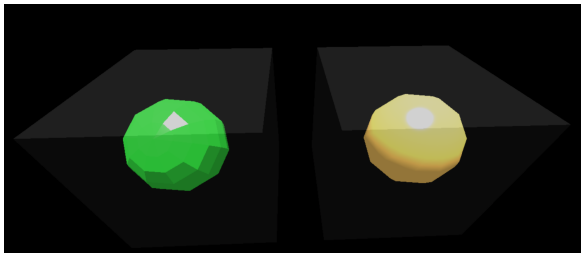

Figure 13 – Two spheres, contained in cubes with similar geometry and material intended to mimic glass.

Transformations (scale and rotation)

In this section some notions about grouping objects and some transformations will be discussed. The corresponding file is Transformations (scale and rotation).htm.
A mesh for a box geometry and 4 meshes for sphere geometries were created (the materials and lighting are not important for this section). All meshes within this scene were grouped within a single object of class Object3D, which is the base class for most objects in Three.js. The Scene object itself is a subclass of Object3D, as is the Mesh object, and Object3D can be used to group objects inside a Scene the same way that a Scene object can contain multiple objects, and is useful to apply transformations and settings to all objects in the group.

```
car = new THREE.Object3D();
car.add(box);
car.add(sphere1);
car.add(sphere2);
car.add(sphere3);
car.add(sphere4);

scene.add(car);
```

In this case the ‘car’ object is the instance of Object3D that contains all other objects (the box and the 4 spheres), which is then added to the scene.
The box geometry was defined as a cube measuring 1 unit on each side:

```
const boxGeometry = new
THREE.BoxGeometry(1, 1, 1);
…
const box = new
THREE.Mesh(boxGeometry,
boxMaterial);
```

Afterwards, a transformation was applied to the mesh for this cube:

```
	box.scale.set(2,1,4);
```

The result is that the coordinates of each vertex were multiplied by different factors depending on the axis, a factor of 2 for the x coordinates, 1 for the the y coordinates and 4 for the the z coordinates, effectively doubling the

dimensions of the cube along the x axis, keeping the same dimensions along the y axis and quadrupling the dimensions along the z axis. The size of the geometry remains the same, but when the geometry is rendered, a transformation is applied to it, in this case a scaling transformation.

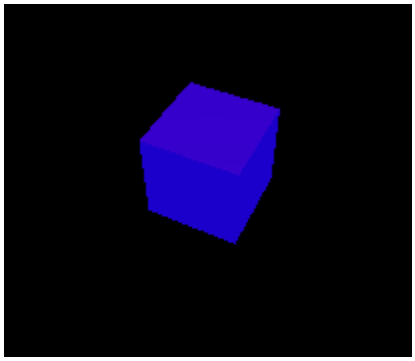

Figure 14 – Cube without scaling.

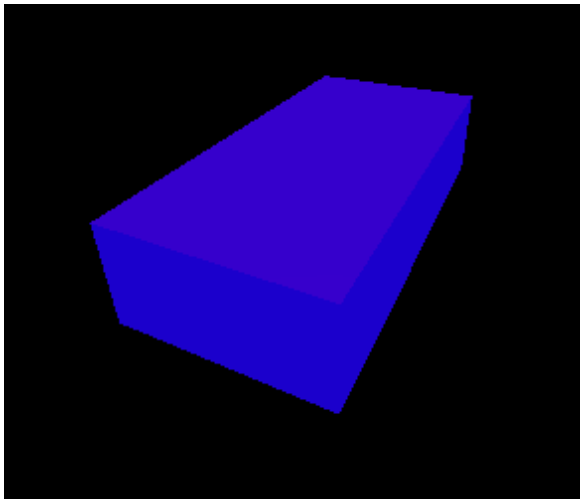

Figure 15 – cube after scaling transformation.

The spheres were then placed at the bottom vertices of the scaled cube, which now forms a parallelepiped, to form a crude “car”:

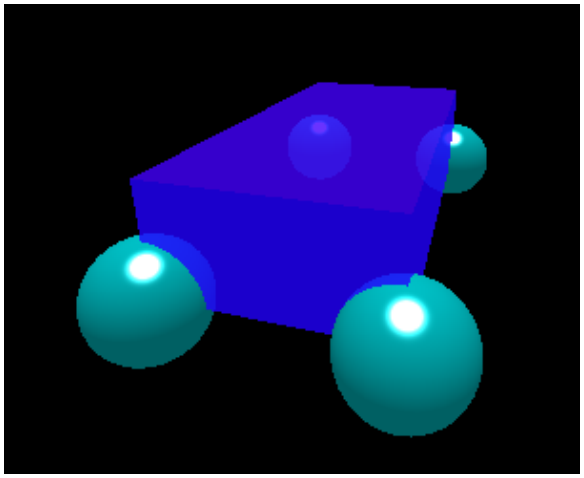

Figure 16 – Final result of the “car”.

Just like translation and rotation transformations, the rotation transformation used here for the cube can be expressed through a 4 by 4 matrix, in accordance with the WebGL standard [1]. In Three.js, 4 by 4 matrices are represented through a Matrix4 object, which serves a similar purpose for matrices of what the Vector3 object does for 3D vectors. This transformation matrix for each object is stored in the Object3D.matrix field, and demonstrate their content was outputted to the console through the following lines:

```
console.log("sphere1
transformation matrix:",
sphere1.matrix);
console.log("box transformation
matrix:", box.matrix);
```

By checking the ‘elements’ field for the objects outputted to the console and rearranging them by columns we can see that for the cube the transformation matrix is defined as:

2, 0, 0, 0
0, 1, 0, 0
0, 0, 4, 0

0, 0, 0, 1

The first 3 lines and columns are what we already know from linear algebra as a scaling transformation, with entries along the diagonal line representing the factor for each dimension, while the extra dimension of the square matrix is used in the WebGL standard to convey extra information and represent translation transformations in the same matrix.

Looking now at the transformation matrix for the sphere1, in position (1, -0.5, 2):

1, 0, 0, 0
0, 1, 0, 0
0, 0, 1, 0
1, -0-5, 2, 1

We can see from the diagonal entries that no scaling transformations were applied to the sphere, and it is evident the use of the extra dimension to express the translation operation that was used to center the sphere on position of the box vertex, at (1, -0.5, 2).

Transformations (rotations).htm

In this section some further notions about how transformations are applied to objects and groups will be discussed. The corresponding file is Transformations (rotations).htm.

Taking the code from the previous section as a starting point, the spheres were replaced with cylinders:

```
const cylinderGeometry = new THREE.CylinderGeometry(0.5, 0.5, 0.2, 40);
...
const cylinder1 = new THREE.Mesh(cylinderGeometry, sphereMaterial);
cylinder1.position.set(1, -0.5, 2);
cylinder1.rotation.z = Math.PI/2;
const cylinder2 = new THREE.Mesh(cylinderGeometry, sphereMaterial);
cylinder2.position.set(1, -0.5, -2);
cylinder2.rotation.z = Math.PI/2;
const cylinder3 = new THREE.Mesh(cylinderGeometry, sphereMaterial);
cylinder3.position.set(-1, -0.5, -2);
cylinder3.rotation.z = Math.PI/2;
const cylinder4 = new THREE.Mesh(cylinderGeometry, sphereMaterial);
cylinder4.position.set(-1, -0.5, 2);
cylinder4.rotation.z = Math.PI/2;
```

Unlike spheres, the orientation of a cylinder is important, so rotation transformations were applied to them to rotate them 90 degrees around the z axis, which were expressed through Euler angles in radians, which Three.js then converts internally to a matrix representation as described in the previous section, (which is placed in the first 3x3 entries of the 4x4 matrix, in accordance with what we know from

linear algebra and the WebGL standard).
Additionally, a representation of the origin of the scene and its axis was also created, with cylinders along the direction of each axis and intersecting at the origin, with red, green and blue for the x, y and z axis, respectively.

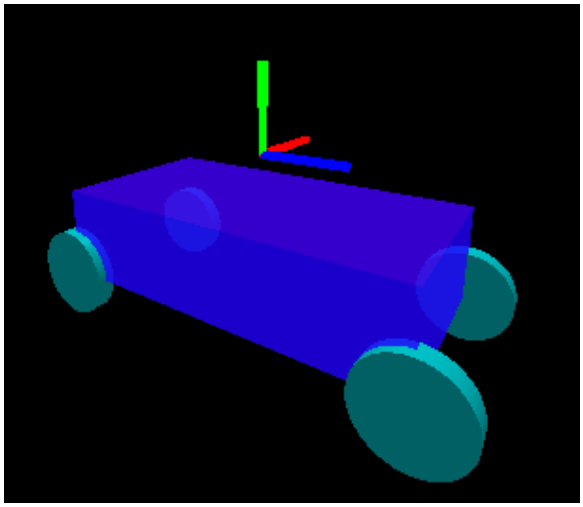

Figure 17 – final “car” with cylinders, with the axis representation above it.

To facilitate the animation of the “car” traveling along a circle with 3 units of radius and centered on the (0,0,-3) point, some preparatory code was written:

```
const carRefPoint = new
THREE.Object3D(),
carPathRadius = 3;
carRefPoint.position.z =
-carPathRadius;
carRefPoint.add(car);
scene.add(carRefPoint);
car.position.x =
-carPathRadius;
car.rotation.x = Math.PI /
2;
```

So the car was placed inside another Object3D object, to be used as the reference point for the rotation. All transformations on Three.js can be relating to the object’s local coordinate system, that is, the position of its parent (the Object3D in which it is contained) or the transformation can be defined with respect to the world coordinate system, which is the scene itself, since it is a subclass of Object3D and contains all objects to be rendered. When the .position, .rotation and .scale properties are set, they define the transformation with relation to its parent, so the rotations of the cylinders previously applied were in relation to their local coordinate system, which was the car, so now that we apply further transformations to the car itself (which contains the cylinders), the relative positions of the cylinders and the box are unchanged.
Finally, the animation itself was defined in the render loop:

```
carRefPoint.rotation.z +=
0.01;
```

The rotation was applied to the reference point of the car, which is the parent of the car object. The car object was translated in relation to this reference point, so as this reference point rotates all objects defined as its children are rotated as well, maintaining their relative positions according to the transformations defined in their local coordinate systems, allowing the car to travel around in a circle with the radius of the distance that the car was

translated in relation to the reference point.

References

[1] - http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices