Lesson 2 - Projections, lighting and transformations

Beatriz Sofia Mesquita Gonçalves, 115367

December 2022

Student Beatriz Sofia Mesquita Gonçalves, 115367
Professors Paulo Dias and Beatriz Sousa Santos

Introduction

This assignment was developed in the context of the Information Visualization course of the Data Science Master's program at the University of Aveiro. To run the code, create a local server, e.g. live server and run in localhost the pages.
This second delivery consists of a brief introduction to projections, lighting and transformations in *three.js*. The **outline** of this assignment is:

- Camera models: perspective and orthogographic cameras;
- Interaction with camera;
- Lighting and shading;
- Transformations.

1 Camera models

We start by modifying the first example from the first lesson to visualize the cube in wireframe [3]. We must activate the appropriate material property. Also disable the cube rotation.

```
// Create the cube
      const cube = new THREE.Mesh(
      new THREE.BoxGeometry(1, 1, 1),
      new THREE.MeshBasicMaterial({ color: 0x00ff00, wireframe: true })
      );
```

Listing 1: Activate the appropiate material property.

Instead of using a perspective camera, we will now use an Orthographic Camera [8]. We will also modify the parameters so that the world view is between -3 and 3 on the x-axis while respecting the window's aspect ratio.

```
// Set the aspect ratio of the window
      let aspectRatio = window.innerWidth / window.innerHeight;

      // Set the x and y limits of the camera's view
      let xLimits = 3;
      let yLimits = xLimits / aspectRatio;

      // Create the camera
      const camera = new THREE.OrthographicCamera(
      -xLimits,  // left
      xLimits,  // right
      -yLimits,  // top
      yLimits,  // bottom
      0.1,      // near
      1000      // far
      );
```

Listing 2: Using the Orthographic Camera.

Lastly, we add a function to ensure that the cube aspect ratio does not change when the window is resized (adapted from the viewport Update in the first lesson):

```
// Add a function to update the camera and cube when the window is resized
    function onWindowResize() {
        // Update the aspect ratio of the window
        aspectRatio = window.innerWidth / window.innerHeight;

        // Update the x and y limits of the camera's view
        xLimits = 3;
        yLimits = xLimits / aspectRatio;

        // Update the camera
        camera.left = -xLimits;
        camera.right = xLimits;
        camera.top = yLimits;
        camera.bottom = -yLimits;
        camera.updateProjectionMatrix();

        // Update the size of the cube
        cube.scale.set(xLimits, yLimits, 1);

        // Update the size of the renderer
        renderer.setSize(window.innerWidth, window.innerHeight);
        }

        // Add an event listener to trigger the function when the window is resized
        window.addEventListener('resize', onWindowResize, false);


        // Create the renderer
        const renderer = new THREE.WebGLRenderer();

        // Set the size of the renderer
        renderer.setSize(window.innerWidth, window.innerHeight);

        // Add the renderer to the DOM
        document.body.appendChild(renderer.domElement);
```

Listing 3: Viewport update.

As an output we get:

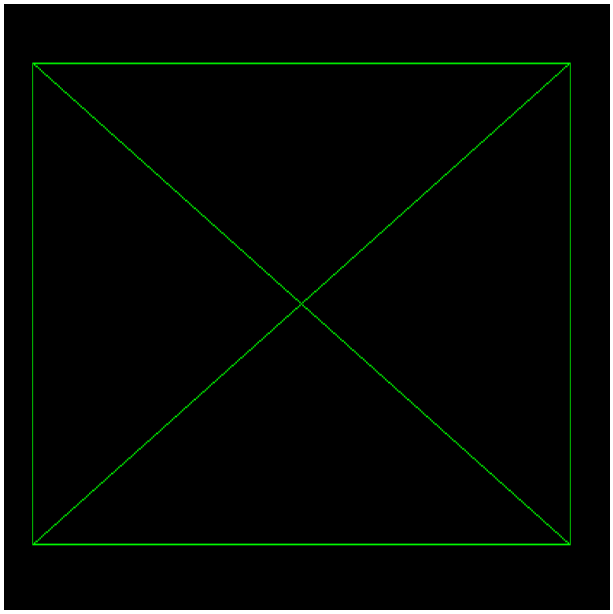

Figure 1: Orthographic Camera output.

We can compare this result from Fig.1 with the output using the perspective camera in wireframe:

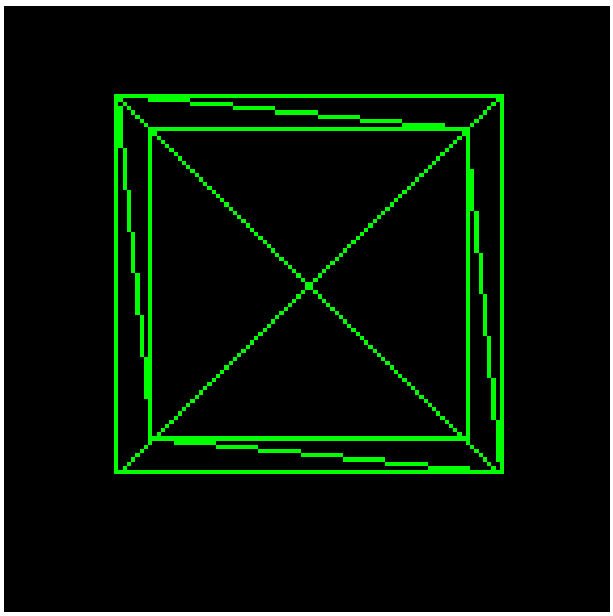

Figure 2: Perspective Camera output.

The main difference between a perspective camera and an orthographic camera is the way they project the 3D scene onto the 2D image plane.

A **perspective camera** uses the field of view angle to create a sense of depth and distance in the image. Objects that are further away from the camera appear smaller, and the lines of the objects converge at a single point on the horizon, called the vanishing point. This creates a more realistic, 3D perspective of the scene. [9]

On the other hand, an **orthographic camera** does not use a field of view angle to project the image. Instead, it projects the scene onto the image plane with parallel lines, resulting in a more flat, 2D representation of the scene. In an orthographic projection, all objects in the scene are the same size, regardless of their distance from the camera. [8]

This way, if we want to create a more realistic, 3D perspective of the scene, we may want to use a perspective camera. If you want to display the scene in a more flat, 2D manner, we may want to use an orthographic camera.

FILE: camera*models.html*

2 Orbit control

The *OrbitControls* class in *Three.js* allows us to control the camera pose using mouse or touch input. We can use it to pan, zoom, and rotate the camera around a central point in the scene. [7] To use these controls we had to include the file separately in your HTML and also add the following script:

```
<script src="https://threejs.org/examples/js/controls/OrbitControls.js"></script>
```

Listing 4: Using OrbitsControl.

However, the page https://threejs.org/examples/js/controls/OrbitControls.js was not available at the time this work was done. Thus, it was not possible for us to get an output for this part of the work. Anyway, we tried to make the code anyway, to show the steps of the process.

We start by creating a line for the camera control:

```
// Create the camera controls
      const controls = new THREE.OrbitControls(camera, renderer.domElement);
```

Listing 5: Camera Control.

We update the camera controls in the render function:

```
// Create a loop to update and render the scene
      function animate() {
        requestAnimationFrame(animate);

        // Update the scene

        // Update the camera controls
        controls.update();

```

```
        // Render the scene
        renderer.render(scene, camera);
      }

      // Start the loop
      animate();
```

Listing 6: Update controls.

FILE: orbit*control.html*

3 Lighting and materials

We will now add lights to the scene. We get back to the perspective camera, disable the wireframe and turn the rotation of the cube back on.

We create a DirectionalLight [2] at position 0.5.0 with color 0xffffff and intensity 1.0. The *DirectionalLight* class in *Three.js* is used to simulate a light source that is infinitely far away and shines in a particular direction. It is useful for creating shadows and illuminating large areas of a scene. [2]

```
// Create the DirectionalLight
      const light = new THREE.DirectionalLight(0xffffff, 1.0);

      // Set the position of the light
      light.position.set(0, 5, 0);

      // Add the light to the scene
      scene.add(light);
```

Listing 7: Directional Light.

We also add an ambient light. An *ambient light* in *Three.js* is a light source that illuminates all objects in the scene equally, regardless of their position or orientation. It is used to create a general, overall lighting for the scene [1]:

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

Listing 8: Ambient Light.

Also, in order for the object to interact with light it is necessary to use a material of a different type from MeshBasicMaterial. So, we will replace the MeshBasicMaterial with a MeshPhongMaterial material:

```
const geometry = new THREE.BoxGeometry( 1, 1, 1 );
      const material = new THREE.MeshPhongMaterial({
              color: '#006063',
        specular: '#a9fcff',
        shininess: 100
          });
      const cube = new THREE.Mesh( geometry, material );
      scene.add( cube );
```

Listing 9: MeshPhongMaterial.

We also highlight some differences between *MeshBasicMaterial* and *MeshPhongMaterial*:

- **Shading**: *MeshBasicMaterial* is a flat shading material, which means that it does not take into account the position and orientation of the light sources in the scene. It always renders the faces of the object with a single color, regardless of the lighting conditions. On the other hand, *MeshPhongMaterial* is a smooth shading material, which means that it takes into account the position and orientation of the light sources in the scene. It interpolates the colors of the faces of the object based on the lighting conditions, creating a more realistic appearance;
- **Lighting**: *MeshBasicMaterial* does not support lighting, which means that it does not respond to light sources in the scene. It always renders the object with a single, constant color. On the other hand, *MeshPhongMaterial* supports lighting and will respond to light sources in the scene, creating highlights and shadows on the object;
- **Reflections**: *MeshBasicMaterial* does not support reflections, which means that it does not reflect the environment or other objects in the scene. On the other hand, *MeshPhongMaterial* supports reflections, and we can specify a texture or a color that the object will use to reflect the environment or other objects in the scene;

- **Performance**: *MeshBasicMaterial* is generally faster to render than *MeshPhongMaterial*, as it does not have to take into account the lighting conditions in the scene. However, *MeshBasicMaterial* may not produce as realistic or visually appealing results as *MeshPhongMaterial* in some cases.

[5] [6]

As an output we get, for the rotating cube:

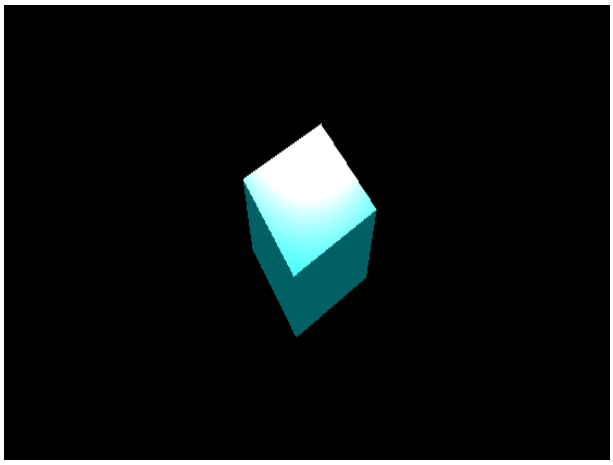

Figure 3: Lighting and Materials' output.

FILE: lighting*andmaterials.html*

4 Shading

Now, we modify the original rotating cube example to represent not a cube but a sphere (primitive sphereGeometry) of radius 1 and modify parameters 2 and 3 with the wireframe option active.

```
1 // Definition of an object/geometry and camera position
2       var sphereGeometry = new THREE.SphereGeometry(1, 32, 32);
3             var sphereMaterial = new THREE.MeshBasicMaterial({color: 0xffffff, wireframe: true
    });
4             var sphere = new THREE.Mesh(sphereGeometry, sphereMaterial);
5       scene.add( sphere );
```

Listing 10: Creating the sphere.

The *widthSegments* and *heightSegments* parameters of the *SphereGeometry* constructor correspond to the number of segments along the width and height of the sphere, respectively.

Each segment is a line that divides the sphere into smaller pieces. The more segments we have, the smoother and more detailed the sphere will be, but it will also require more processing power to render.

For example, if we set *widthSegments* to 32 and *heightSegments* to 32, the sphere will be divided into 32 segments along the width and 32 segments along the height, for a total of 1024 segments. This will create a relatively smooth and detailed sphere. [10]

As an output we get, for the rotating sphere:

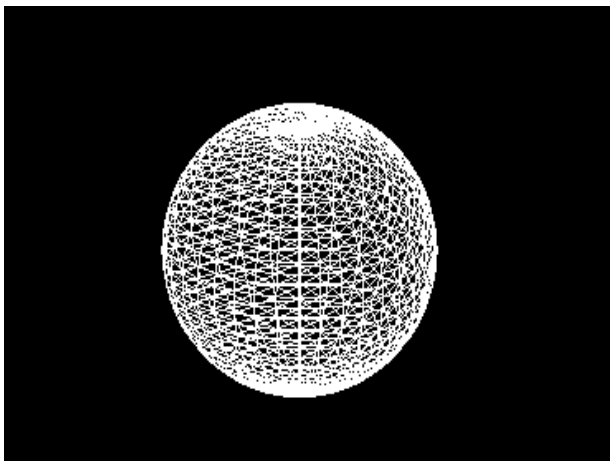

Figure 4: Wireframe sphere with 32 segments.

FILE: shading$wireframetrue.html$
We disable wireframe and set the number of segments to 10, obtaining the output:

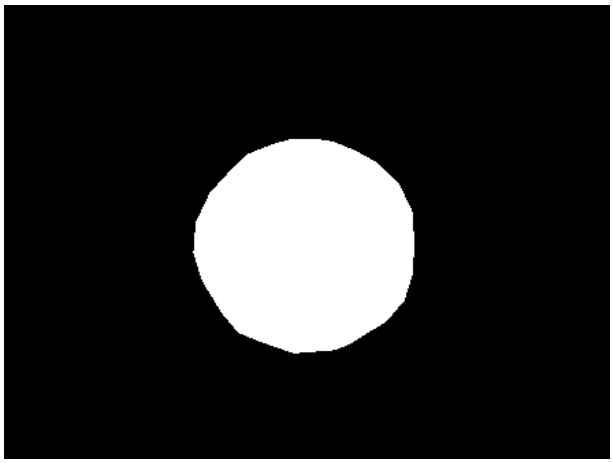

Figure 5: Sphere with 10 segments.

FILE: shading$nowireframe.html$
Now we create another sphere with the same characteristics and place one sphere at x=-2.5 and the other at x=2.5, using the position.x method.

```
// Definition of an object/geometry and camera position
            var sphereGeometry = new THREE.SphereGeometry(1, 10, 10);
            var sphereMaterial = new THREE.MeshBasicMaterial({color: 0xffffff});

            var sphere1 = new THREE.Mesh(sphereGeometry, sphereMaterial);
            sphere1.position.x = -2.5;

            var sphere2 = new THREE.Mesh(sphereGeometry, sphereMaterial);
            sphere2.position.x = 2.5;
      scene.add( sphere1 );
            scene.add( sphere2 );
```

Listing 11: Creating another sphere.

And we get:

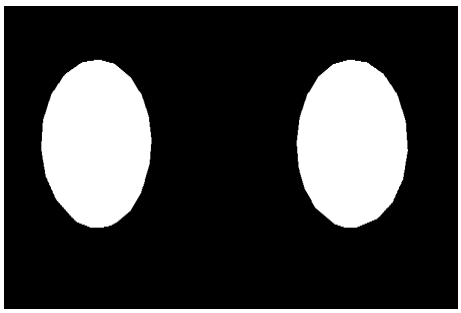

Figure 6: Two spheres at defined positions.

FILE: shading*nowireframe2spheres.html*

We now add the ambient light and directional light from the previous exercise located between the two spheres with y=5 and get:

```
// create an ambient light
            var ambientLight = new THREE.AmbientLight(0x404040);

            // create a directional light
            var directionalLight = new THREE.DirectionalLight(0xffffff, 0.5);
            directionalLight.position.set(0, 5, 0);

            // add the lights to the scene
            scene.add(ambientLight);
            scene.add(directionalLight);
```

Listing 12: Creating ambient and directional lights.

Also, we apply the same MeshPhongMaterial to the two spheres and modify the flatShading option of one of the materials by toggling between true and false and observe the result.

```
// create a MeshPhongMaterial with a green color and flat shading
            var sphereMaterial1 = new THREE.MeshPhongMaterial({color: 0x00ff00, flatShading:
    true});

            // create a MeshPhongMaterial with a blue color and smooth shading
            var sphereMaterial2 = new THREE.MeshPhongMaterial({color: 0x0000ff, flatShading:
    false});
```

Listing 13: Flat Shading.

The result is:

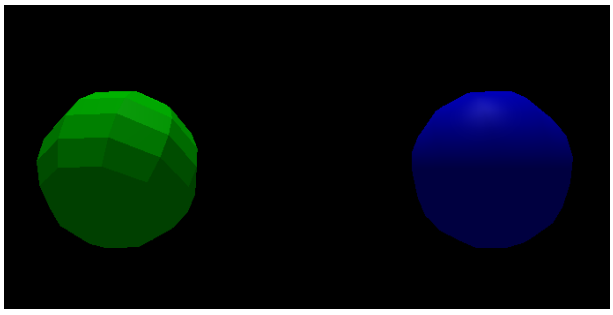

Figure 7: Two spheres at defined positions, light and flat shading.

FILE: shading*nowireframelighting.html*

Now, we apply to the first sphere a *MeshLambertMaterial* type material with the same characteristics as sphere 2. In the Lambertian-type material, we remove the specular and shininess components. It will appear to scatter light evenly in all directions, as is characteristic of Lambertian materials. This can give the first sphere a matte, diffuse appearance, as opposed to the shiny, specular appearance that you may have observed with the original material. [4]

So, we get:

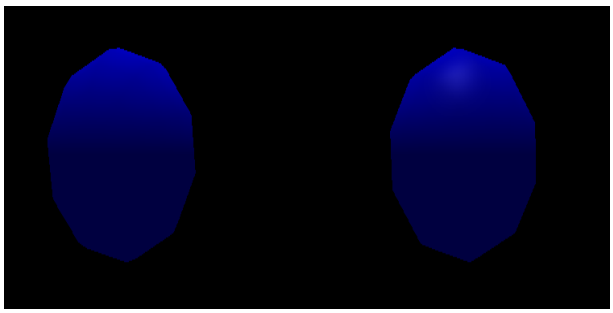

Figure 8: Lambertian Materials.

FILE: shading*optional*1.*html*

The, we modify the properties of the spheres by selecting some values from the table on the notes to see the effects of different materials. We chose the values for emerald, gold and silver:

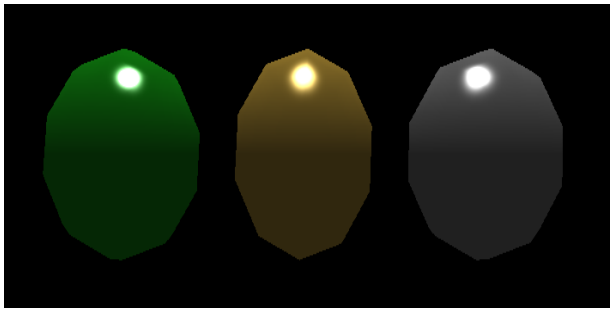

Figure 9: Emerald, Silver and Gold Spheres.

FILE: shading*changingproperties.html*

To finish, we add the following lights, all pointing to the origin of the scene: red directional light in position (-5,0,0), blue directional light in position (5,0,0) and green spotlight light in position (0,0,-5) with angle Math.PI/20 and target object in (-2.5,0,0), getting as output, for the three rotating spheres:

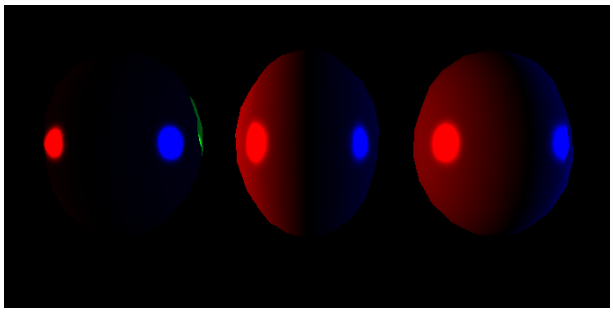

Figure 10: Three spheres with specific lights.

FILE: shading*optional*2.*html*

5 Transparency

In the previous example (two spheres located at x=-2.5 and x=2.5), we will add cubes with a slightly larger size around original spheres, Modifying the opacity parameter to adjust the transparency and using the given material for these two models. We observe the effect:

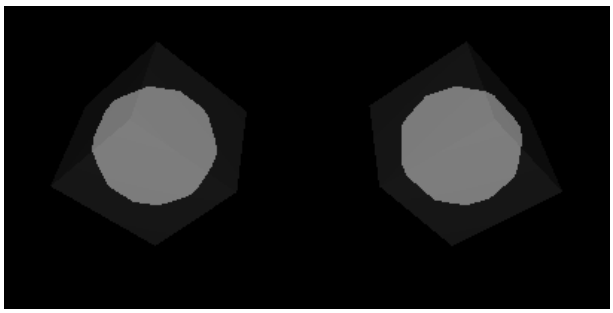

Figure 11: Transparency.

FILE: transparency.html

6 Transformations (scale and rotation)

We will create a new scene consisting of a box of size (2,1,4) (use the scale property) at position (0,0,0) and four spheres (radius 0.5) centered on its lower vertices (like a little car). Instead of adding multiple separate meshes, we will add multiple meshes into a single THREE.Object3D() via the add command.

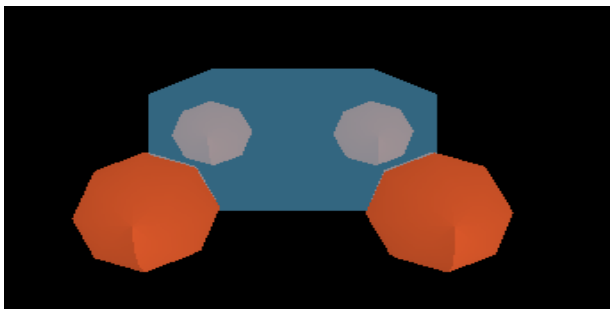

Figure 12: Car.

To view the transformation matrices of the parallelepiped and of one of the spheres on the console, we can access the matrix property of the objects:

```
1 console.log(box.matrix);
2 console.log(sphere1.matrix);
```

Listing 14: Transformation Matrices.

FILE: transformations*scaleandrotation.html*

7 Transformations (rotations)

To finish this assignment, we create an object that represents a coordinate system using three red, green, and blue cylinders (CylinderGeometry) for each axis. The three cylinders must belong to a single object named axis. We then add this object to the previous scene and replace the spheres with cylinders with radius 0.5 and height 0.2. We finish by adding an animation to move the "car" along a predefined circuit: perform a rotation of radius 1 around the point (0,0,-1).

We get the output:

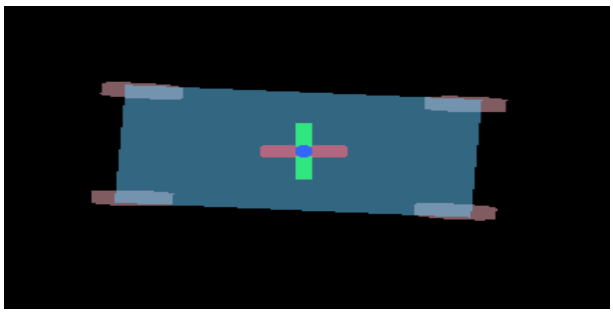

Figure 13: Transformations (rotations); Car2.

We use this camera position so we can perceive well the circuit it does.
FILE: transformations*rotations.html*
[11]

References

[1] Ambientlight. `https://threejs.org/docs/#api/lights/AmbientLight`. Accessed: 2022-12-21.

[2] Directionallight. `https://threejs.org/docs/#api/en/lights/DirectionalLight`. Accessed: 2022-12-21.

[3] First example: Rotating cube. `https://threejs.org/docs/index.html#manual/introduction/Creating-a-scene`. Accessed: 2022-12-21.

[4] Lambertian reflectance and linear subspaces. `https://www.researchgate.net/publication/3906163_Lambertian_Reflectance_and_Linear_Subspaces`. Accessed: 2022-12-21.

[5] Meshbasicmaterial. `https://threejs.org/docs/#api/materials/MeshBasicMaterial`. Accessed: 2022-12-21.

[6] Meshphongmaterial. `https://threejs.org/docs/#api/materials/MeshPhongMaterial`. Accessed: 2022-12-21.

[7] Orbitscontrol example. `https://threejs.org/docs/#examples/en/controls/OrbitControls`. Accessed: 2022-12-21.

[8] Orthographic camera. `https://threejs.org/docs/#api/en/cameras/OrthographicCamera`. Accessed: 2022-12-21.

[9] Perspective camera. `https://threejs.org/docs/#api/en/cameras/PerspectiveCamera`. Accessed: 2022-12-21.

[10] Sphere geometry. `https://threejs.org/docs/#api/geometries/SphereGeometry`. Accessed: 2022-12-21.

[11] uainfovis/three.js/lesson01. `https://github.com/pmdjdias/ua_infovis/tree/master/Three.js/Lesson_01`. Accessed: 2022-12-14.