Interactive Graphics - Final Project

Solar System

A.Y. 2018/2019

Gianmarco Cariggi 1698481
Marco Costa 1691388

12th July 2019

1 Introduction

For the final project of the course of Interactive Graphics we decided to make a simulator of our solar system, trying to reproduce it in the most faithful and realistic way possible, using of course assumptions and approximations to simplify its implementation.

1.1 Requirements

- Hierarchical models
- Lights
- Textures
- User interaction
- Animation

1.2 Environment

The project is developed in WebGL (Web Graphics Library). It's a JavaScript API for rendering interactive 2D and 3D graphics within any compatible web browser without the use of plug-ins. It is fully integrated with other web standards, allowing GPU-accelerated usage of physics and image processing and effects as part of the web page canvas.
Click here to check if your browser supports WebGL.

1.3 Libraries not developed by the team

- **Three.js** (rev. 107dev): 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*.
- **Materialize** (v. 1.0.0): a design language that combines the classic principles of successful design along with innovation and technology.
- **jQuery** (v. 3.4.1): it takes a lot of common tasks that require many lines of JavaScript code to accomplish, and wraps them into methods that you can call with a single line of code. It also simplifies a lot of the complicated things from *JavaScript*, like *AJAX* calls and *DOM* manipulation.

1.4 Assumptions and approximations

As is well known, the planets follow an elliptical orbit (a "flattened" circumference), and the sun occupies one of the two focuses. Rotating along this orbit, we have the 4 seasons (also due to the inclination of the axis). For simplicity of implementation and positioning of the various planets, we have approximated the orbit with a circumference, and the sun occupies centre.

Another important fact is that the orbit of all the planets varies continuously (for example, the orbit of the earth is not the same from one year to the next), this is because there is the gravitational influence of all the celestial bodies present in the universe (and in particular in the solar system, being closer). In the solar system, besides the sun, Jupiter has an important influence, being the biggest planet. In our simulator we do not take into account all this, the planets revolve around the sun with the same orbit (we can say that the sun is the only body that influences the gravitation of the planets).

The last detail is the actual position of the planets. Probably they will not be located in the real position, because in our simulator this position depends on the date, and in JavaScript the function **Date.getTime()** returns the milliseconds passed from January 1, 1970 to the moment in which the function is called. In theory we should have the milliseconds passed from the birth of the solar system to today, or set a certain initial position so as to match the real position of each planet with that in the simulator. We decided to approximate the initial position by consulting an online simulator of the solar system[1], setting the date to January 1, 1970. The position of each planet is determined by the angle formed by the straight line that connects each of the planets to the sun. Taking this into account, we have added an initial angle for each planet, obtained by comparing the position assumed by the various planets in our simulator and that of TheSkyLive. Obviously this angle is a simple approximation, because as said before the planets have an elliptical orbit, while in the simulator follow a circular orbit. The more time passes, the more the position of the planets in the simulator could be different from the real one. Looking at Pluto, it is possible

to notice this easily, because this planet follows the most particular orbit. It must be noted however that every planet makes a turn around the sun in the real time of revolution and makes a turn around its axis (inclined with respect to the equatorial plane) in the real time of rotation (for example the earth takes exactly one year to make the revolution around the sun and 24 hours to make a turn on itself).

2 Usage

The graphic interface allows to user to interact with our projects. First of all there is a fast loader page meanwhile the browser loads all the data.

At the beginning camera is focused on Sun and the *follow-planet* flag is enabled, then you can only move camera around the sun. In order to move in the solar system, you have to disable it through the main menu.
In meanwhile a background music starts in order to create a pleasant atmosphere and to take a good travel around the solar system. If you don't like it, you can disable through the specific button at bottom right of the window.
Since you disable the *follow-planet*, now you can move.
Have a good trip!

2.1 Mouse Controls

The mouse is very important to fully enjoy this experience, so we explain the available mouse controls below:

- **Left Click**: Normally you click button or elements in the menu. If menu is opened and click on the solar system, it will close. If you click and drag the camera orbits around the target point.
- **Double Left Click**: If it's done over a celestial objects (except *Asteroid Belt*), it permits to select and focus the camera on it.
- **Right Click**: If the *follow-planet* flag is disabled, you can move around the environment clicking and dragging.
- **Scroll Wheel**: As usual, the scroll wheel permits to zoom in or zoom out on camera's target.
- **Move on the object**: Moving over the planets, a pop-up window appears where there are all planet's physical data.

All the commands are possible through the using of *OrbitControls* of *Three.js*.

2.2 Menu Options

The menu offers to you a lot of options to custom the animation and the aspect of solar system. It can be opened or closed simply clicking on the relative button in the top right of the screen. So let to describe all the options:

- **Play / Pause Animation**: Play or pause all animations of the solar system. (Default: Play)
- **Rotation**: Play or pause only the rotation on the axis of planet. (Default: Play)
- **Revolution**: Play or pause only the revolution orbit. (Default: Play)
- **Trajectories**: Show / Hide the orbit trajectories. (Default: ON)
- **Inclined orbit**: Set / Unset the real inclination of the planet's orbit. (Default: ON)
- **Date**: Set a specific date to see the state of solar. system.
- **Time**: Set a specific time to see the state of solar system.
- **Speed**: Set the speed of time flowing (and then the animation). (Default: 1.0x)
- **Far**: Set the far parameter of camera. (Default: 10000).
- **Camera**: Set the camera target planet (as double click on a planet). (Default: Sun)
- **Follow**: Camera follows or not the target planet. (Default: ON)
- **Rotate Camera**: Camera orbits around the target planet. (Default: OFF)
- **Earth clouds**: Show / Hide the Earth's clouds. (Default: OFF)
- **Asteroid Belt**: Show / Hide the asteroid belt. (Default: OFF)
- **Ambient light**: Turn on / off the ambient light. (Default: OFF)
- **Sun light**: Turn on / off the Sun light. (Default: ON)
- **Sun glow**: Turn on / off the Sun glow. (Default: ON)
- **Sun light intensity**: Set the sun light intensity (Default: 1.5)
- **Wallpaper**: Choose the background.
- **Track**: Choose the background music.
- **Volume**: Set the volume of track. (Default: 0.3)

3 Implementation

In this section, let we how we implemented all the features in the project. We prefer to divide this section in subsection to make easier the lecture and understanding.

3.1 Three.js basic components

The first part is about the initialization of Three.js, in this case we create the scene and the other fundamental elements for a WebGL environment. We do this in the *init* function.

3.1.1 Scene

The first element for Three.js environment is the *scene*[5]. It is rendered and be simply instantiate using:

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

3.1.2 Camera and OrbitControls

The second step is the camera, it allows to watch the scene. In our case, we use a perspective camera, with a $FOV = 45°$, $near = 0.1$ and an adapting *far* to make visible the far objects[5].

Beside the camera, we use OrbitControls library that allows mouse interactions with the camera. The parameter passed to the function specify which area of the canvas is under control of *OrbitControls*. We choose the container, in this way, when the menu is opened, menu area is ignored by the controller[5].

```
1 camera = new THREE.PerspectiveCamera(45, window.innerWidth / window
      .innerHeight, 0.1, far);
2 camera.position.y = 45;
3 camera.position.z = 100;
4 camera.lookAt(new THREE.Vector3(0, 0, 0));
5 controls = new THREE.OrbitControls(camera, document.getElementById(
      "container"));
6 controls.autoRotateSpeed = 1;
```

3.1.3 Raycaster

Raycaster is very important in our project, because we have to establish which object is "captured" by the mouse cursor for different goals as we explain later[5]. It's sufficient an instruction:

```
1 raycaster = new THREE.Raycaster();
```

3.1.4 Renderer

We use the standard *WebGLRenderer*[5], enabling the antialiasing filter and using Percentage-Closer Soft Shadows (PCSS) as shadow maps filter[6].

```
renderer = new THREE.WebGLRenderer({
antialias: true
});
renderer.setSize(window.innerWidth, window.innerHeight);
document.getElementById("container").appendChild(renderer.
    domElement);
renderer.shadowMap.enabled = true;
renderer.shadowMap.type = THREE.PCFSoftShadowMap;
```

3.1.5 Texture Loader

Texture loader[5] is used to load all textures to relative objects.

```
textureLoader = new THREE.TextureLoader();
```

3.2 Light

In our project, light is a fundamental element, all the objects in solar system receive the sunlight. We use two kind of light source: point light coming from the sun and an optional ambient light for a better view of the shaded parts.

3.2.1 Point Light

Point light represents the light Sun produces in its core using the energy released through the nuclear fusion of hydrogen atoms[7].

The implementation of a point light source requires the use of Three.js library that makes it easier[5]. The code is contained in *createSun()* function:

```
sunLight = new THREE.PointLight("rgb(255, 220, 180)", 1.5);
sunLight.castShadow = true;
sunLight.shadow.bias = 0.0001;
sunLight.shadow.mapSize.width = 2048;
sunLight.shadow.mapSize.height = 2048;
scene.add(sunLight);
```

First of all we set the colour of light and its intensity. In the next lines, instead, we put the shadow parameters (bias and map size of shadow). Finally we add the light to Three.js scene.

3.2.2 Ambient Light

We add the ambient light to show better the shadow areas. For this reason, you can turn off with relative switch and at the start it's off. As for the point light, also the ambient light is implemented using the specific Three.js function[5]. The code is contained in *init()* function:

```
ambientLight = new THREE.AmbientLight(0xaaaaaa,
    ambientLightIntensity);
scene.add(ambientLight); // Just for few light
```

Simply, we create the ambient light and add it to the scene. The *ambientLight-Intensity* var is used to is used to adjust the intensity of the ambient light via the dedicated slider (it is initially set so that the ambient light is off).

3.3 Music

Music is another very important element in our project, due to amplify the user experience. Also music can be managed through Three.js library.

The first step is to introduce the audio listener which allows the playback of tracks. It needs that AudioListener[5] object is added to camera as you see below:

```
audioListener = new THREE.AudioListener();
camera.add(audioListener);
```

Then, it needs to add the different track and we do this in *ambientMusic()* function:

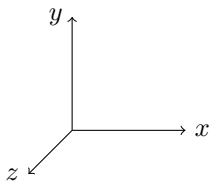
```
for (let i = 0; i < tracks.length; i++) {
  // Create tracks and put it in global audio source array
  sounds[i] = new THREE.Audio(audioListener);

  audioLoader.load(tracks[i], function(buffer) {
    sounds[i].setBuffer(buffer);
    sounds[i].setLoop(true);
    sounds[i].setVolume(volume);
    // Start firts track
    if (i == 0) {
      sound = sounds[0];
      sound.play();
    }
  });
}
```

For every track, it needs to build Audio[5] object that permits to play music, using *AudioListener*. Then we bind the track with Audio object, setting volume and loop playing. Finally if it is the first track, we play it.

3.4 Models

The models in our project are represented by the planets, Sun, Moon and other celestial objects as Asteroid Belt and planet rings. All this object are children of THREE Group Solar System.
All the models and the objects are represented in a 3D space with this axis:



Now let we see all the models in this project.

3.4.1 Planet

The planet is the most common model in our project. It is a sphere and is created in *createPlanet()* function. All needed data to build the planet is taken by Wikipedia articles[3].

```
1 let planetGeometry = new THREE.SphereGeometry(data[Id].size,
      planetSegments, planetSegments);
2 let planetMaterial = new THREE.MeshPhongMaterial({
3   map: textureLoader.load(data[Id].color) // Color texture
4 });
5 //...
6 celestialObjects[Id] = new THREE.Mesh(planetGeometry,
      planetMaterial);
7 //...
8 createOrbit(Id);
```

First of all, we need for a Geometry and Material objects to create a planet. The geometry we chosen is naturally a *SphereGeometry*[5] with as radius a proportion of its real radius. In particular we choose as Earth radius the value of 1, so to find all the other measure we have made the ratio (real planet radius)/(real Earth radius), to find "how many planet is in Earth". In this way we obtain an approximated scale.

Regards to material object, we choose *MeshPhongMaterial*[5], in which we load all the possible texture. In addiction of this, if it's available we load also *bump, specular* and *normal* texture, all taken in this website [2]. More are the available textures and more the planet is realistic.

If the planet has got rings, (Saturn or Uranus), we also create the ring but especially a *THREE Group* so that they make up a system. We had some difficulty loading the texture of the rings, as it was not loaded radially. To solve this problem, we redefined the RingGeometry function, using the one in the *THREEx.planets.js*[**?**] library. This function can be found in the file *helper.js*. Finally we create its orbit trajectory through *createOrbit* function.

3.4.2 Sun

In our implementation Sun can be associated to a planet, also if we wrote a specific function to build it (*createSun()*). Indeed the building of it is very similar to a planet, the only elements that differentiate it is the light, which I have already explained in the previous section, and its Corona that let we describe below:

```
let spriteMaterial = new THREE.SpriteMaterial({
map: textureLoader.load(data[sunId].glow), // Glow texture
color: 0xffffee,
transparent: true,
blending: THREE.AdditiveBlending
});
celestialObjects[sunGlowId] = new THREE.Sprite(spriteMaterial);
celestialObjects[sunGlowId].name = "Sun Glow";
celestialObjects[sunGlowId].scale.set(70, 70, 1.0);
celestialObjects[sunId].add(celestialObjects[sunGlowId]); // This
    centers the glow at the sun.
```

In order to create an effect similar to an atmosphere, we used the Three.js *Sprite*[5] object. It only needs for Material, in particular for *SpriteMaterial*[5]. Thus, after scaling, we add it to sun object.

N.B. The size of the sun is not exactly to scale like the rest of the celestial objects, as its actual size is about 110 times that of the earth. By creating the exact scale size the rest of the planets would have been tiny, so we decided to simply make it bigger than Jupiter.

3.4.3 Orbit Trajectory

Another model in our project is the orbit trajectory of the planets and other celestial bodies. We decide to insert it to show them better to user movements of the bodies and to recognize easier which planet is, the implementation is in *createOrbit()* function.

In order to implement this object, we take inspiration from a question posted in Stack Overflow[8]:

```
let orbitGeometry = new THREE.CircleGeometry(data[Id].distance,
    orbitSegments);
orbitGeometry.vertices.shift(); // Remove center vertex
let orbitMaterial = new THREE.LineBasicMaterial({
  color: 0xffffff
});
orbits[Id] = new THREE.LineLoop(orbitGeometry, orbitMaterial);
```

In this case, we used *CircleGeometry*[5], *LineBasicMateria*[5] and *LineLoop*[5] object. But the most interesting part is the instruction at **line 2** that delete the vertex at centre and so that close the circumference.

3.4.4 Asteroid Belt

In the implementation of the asteroid belt we have encountered many difficulties. In the first version, there was the use of very small spheres, using Sphere Three.js. This choice, however, involved a great use of resources given the large number of objects to be rendered. So we decided to opt for the *Points*[5] object of Three.js. In this way there is a great limiting of resources and the ability to insert a large number of bodies.

The next difficulty was in debugging an error in the code, so the question posted on Stack Overflow was very helpful[9]. The cause is an existent issue

opened in GitHub with *morphAttributes* object, so simply it needs to assign to this object an empty array {}. Let we see the code:

```
//...
let torusAsteroid = new THREE.Mesh(new THREE.TorusGeometry(distance
    , maxOffsetY, planetSegments, planetSegments), new THREE.
    MeshBasicMaterial({
  transparent: true,  // Make torus transparent
  opacity: 0.0        // Make torus transparent
}));
//...
for (let i = 0; i < asteroidCount; i++) {
  // Create a particle with random position
  let asteroidDistance = distance + THREE.Math.randFloat(
      minOffsetXZ, maxOffsetXZ);
  let angle = THREE.Math.randFloat(0, THREE.Math.degToRad(360));
  let coord = new THREE.Vector3();

  coord.x = Math.cos(angle) * asteroidDistance;
  coord.y = THREE.Math.randFloat(minOffsetY, maxOffsetY);
  coord.z = Math.sin(angle) * asteroidDistance;

  // Insert asteroid
  asteroidsGeometry.vertices.push(coord);
}
asteroidsGeometry.morphAttributes = {};     // Used to fix
    updateMorphAttribute bug
celestialObjects[asteroidBeltId] = new THREE.Points(
    asteroidsGeometry, new THREE.PointsMaterial({ size:
    asteroidSize }));
//...
```

In the first step, we implement a *TorusGeometry*[5] object that has the same size of asteroid belt, since in this way if the mouse is over it, *Ray Caster* finds the asteroid belt. For this reason, the torus is invisible and has only this aim.

In the second part, we create every asteroid with a random position such that it is inside the torus volume, adding it as a vertex in the geometry object. Then together the *PointsMaterial*[5], we create the final object.

3.4.5 Rings of planet

As far as the implementation of the planetary rings is concerned, we had some difficulties with the loading of the texture. The problem was that the texture was not loaded radially along the rings. But thanks to the solution found on GitHub[4], we were able to solve the issue, due to a problem related to the object *RingGeometry*[5].

```
let ringGeometry = new THREE.BufferGeometry().fromGeometry(new
    _RingGeometry(data[Id].ringInnerDiameter, data[Id].
    ringOuterDiameter, data[Id].ringSegments));
let ringMaterial = new THREE.MeshPhongMaterial({
  map: textureLoader.load(data[Id].ringColor), // Color texture
  alphaMap: textureLoader.load(data[Id].ringPattern), // Pattern
      texture
  side: THREE.DoubleSide,
```

```
6     shadowSide: THREE.DoubleSide,
7     transparent: true,
8     opacity: 0.8
9   });
10  celestialObjects[data[Id].ringId] = new THREE.Mesh(ringGeometry,
        ringMaterial);
```

Using *_RingGeometry* we manage to load texture in radial way. As other models, it needed to create both the geometry and the material.

3.4.6 Earth cloud

So we decided to add clouds to the Earth to make it more realistic. The problem we encountered was similar to the previous one for the rings of the planets, we found the solution from the same source[5].

The model used is simply a *SphereGeometry*[5], but the difficulty was in creating the material for the sphere. In order to create the material for the model, we decided to use the auxiliary function *createEarthCloudMaterial()*.

```
1  let earthCloudGeometry = new THREE.SphereGeometry(data[earthId].
       size, planetSegments, planetSegments);
2  let earthCloudMaterial = createEarthCloudMaterial();
3  celestialObjects[earthCloudId] = new THREE.Mesh(earthCloudGeometry,
        earthCloudMaterial);
4  celestialObjects[earthCloudId].scale.set(1.02, 1.02, 1.02);
5  //...
```

3.5 Animation

The animation of our solar system is the most important part of our project. In fact, the movement of the planets and their orbits around the sun is the effect that most affects the user and that allows the best to simulate the position of the planets as time passes.

All movements are allowed thanks to the *render* function that allows frame after frame to render the positions of the various celestial bodies in their positions.
The function also allows the room managed by OrbitControls to update its position, using the *controls.update()* native method[5].
We will then analyse all the animations present in our project, in order to explain how they work.
As far as the rotation of celestial bodies is concerned, this is dealt with by the *moveplanet()* function. This function calls for each body, the function for rotation and revolution motion around their orbit centre.

3.5.1 Rotation around own axis

The first animation implemented was the one about the rotation of bodies around their own axis. This animation is done through the *rotationMovement()* function, as below:

```
let theta = THREE.Math.degToRad(360) * (date.getTime() / (data[Id].
    rotationRate * 3600000)); // Cast rotationRate in milliseconds

celestialObjects[Id].rotation.y = theta;
if (Id == earthId) celestialObjects[earthCloudId].rotation.y =
    theta/2;
if (data[Id].hasOwnProperty("ringId")) celestialObjects[data[Id].
    ringId].rotation.z = theta;
```

The theta angle for rotation therefore takes care both the speed of rotation and puts it together with the time variable to establish at what "time of day" the body is, normalizing everything.
Once calculated how much the body must rotate (the theta angle), the rotation is then applied and if it has rings, the geometric transformation is also applied to them.

3.5.2 Revolution orbit

3.6 User Interactions

3.7 HTML and CSS

References

[1] Verify the real-time position of solar system objects.

[2] Planet pixel emporium, website for planet textures.

[3] Wikipedia - for data and descriptions of the celestial objects.

[4] Threex.planets.js - for loading earth clouds texture and have a ring geometry with a radius texture.

[5] ThreeJS documentation: Scene Perspective Camera OrbitControls Raycaster WebGLRenderer. Texture Loader Point Light Ambient Light Audio Listener Audio Sphere Geometry MeshPhongMaterial Sprite Material Sprite Circle Geometry Line Basic Material Line Loop Points Torus Geometry Points Material Ring Geometry

[6] Percentage-Closer Soft Shadows, Randima Fernando, NVIDIA Corporation.

[7] Solar core, Wikipedia article.

[8] Draw a circle (not shaded) using Three.js.

[9] TypeError Stack Overflow question.