Interactive Graphic
Project: Development of a video-game with Three.js

Andrea Macrì
1811286

September 20, 2018

Contents

Chapter 1

Introduction

Three.js is an API that allow users to create, animate and display 3D computer graphics objects or scenes in a web browser. It's practically an high-level library of WebGL, written in JavaScript.

The purpose of this project is to use Three.js to develop a browser video game, focusing on the graphic component of the game and trying to exploit the most important features and methods of this library, such as hierarchical modeling, textures, lightnings, animation and so on.

Chapter 2

Implementation

2.1 Scene and rendering

The main element of a three.js project is the scene object: everything that should be displayed should be added to it.

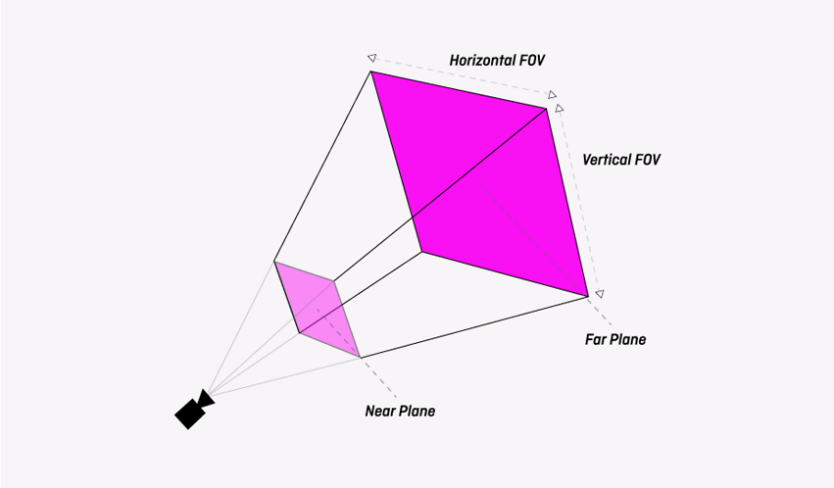
The second important element is the render: it draws everything from the scene to the canvas. To enable animation it's necessary to add an update function, with a recursive call to itself using *renderAnimationFrame* to generate a render loop: every operation defined in this function is recalculated at each iteration.

2.1.1 camera

The implemented camera uses perspective projection. This means that the representation of the scene it's an approximation of a 3D environment on a flat surface, positioned in front of the scene, as it is seen by an eye. So objects are scaled relative to that viewer. The parameters requested by this type of projection are:

- near plane: the distance between the viewer and the plane in front of the scene;
- far plane: the distance between the viewer and the plane behind the scene (farthest field of view possible);
- fov: the angle that can be seen from where the camera is;
- aspect: the ratio between width and height of the plane.

This camera is so positioned above the main planet, on the positive z axis, looking at the negative of the same.



Representation of perspective projection

2.1.2 lighting

There are presents two type of light sources in the scene, both of them set with parameters for the colors and intensity:

- an *Hemisphere Light*: a particular source that cannot cast shadows, positioned directly above the scene, with color fading from the sky color to the ground color;
- a *Directional Light*: this type of source is often used to simulate the sun light, because it acts as if it's really far away from the objects and emits parallel rays in a specific direction. It's positioned in the top right corner of the scene, and differently from the first light source, is able to cast shadows.

It's also present a fog effect, that grows exponentially denser with the distance, in order to simulate depth and show a horizon. The colour of the fog is important for the illusion to work properly and depends on the colour of the lighting.

2.1.3 HUD

The HUD consist of 9 different elements, build as 'div' elements and added as child nodes to the body. Using the 'style' methods it's possible to manage all the features of the text, like colors, fonts, dimension, position in the window (absolute or relative) and so on. They are created within the function *addHUD*, called during the creation of the scene because the most part of them are statics. The changing texts are declared as global variable in order to allow updates. The different elements are:

- Title;
- Difficulty level menu;
- Healt label;

- Healt bar;
- Fuel label;
- Fuel bar;
- Score;
- Start message;
- Game over message.

The title and the two labels are statics. Scores are updates during the all game. The menu is composed by 3 buttons elements, each of them related to a listener that execute a function to set the different difficulty level. The bars are rectangular areas and their widths are directly proportional to the nominal value they represent and are update only when that values are involved. The start text is the one that appear during the first phase, then it becomes a countdown during the positioning of the spaceship (the timing is actually given by the different phases of this animation) until it disappears when the game starts (along with the title and the menu). The game over message, obviously, appears only when the game ends and it's composed by a black horizontal band at the center of the screen with a big red message on it.

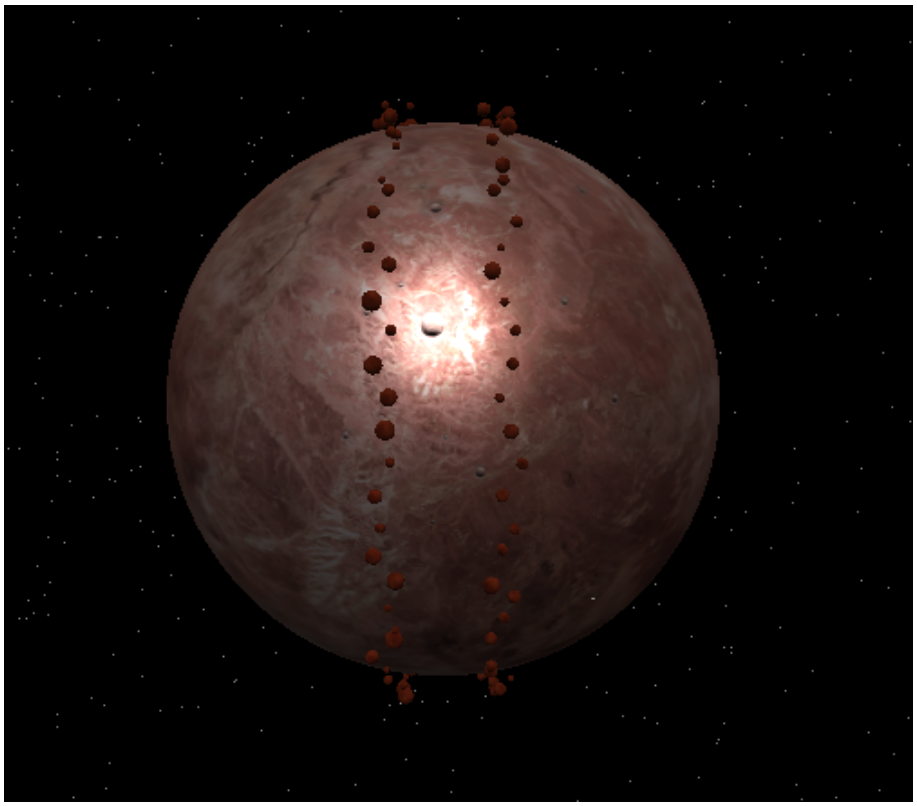
2.2 Create the models

In *Three.js* objects are created using the function *THREE.mesh*, that requires as parameters the geometry and the material features. *Three.js* provides also a multitude of different base geometrical elements and material styles.

There are two main models in this game: the planets and the spaceship, both of them build in a hierarchical way. Hierarchical modeling means the presence of a central unit that works as parent node and of several unites as child nodes, that are linked to the first (or to another higher-level node) and whose positions and animation are expressed in relation to the parent node.

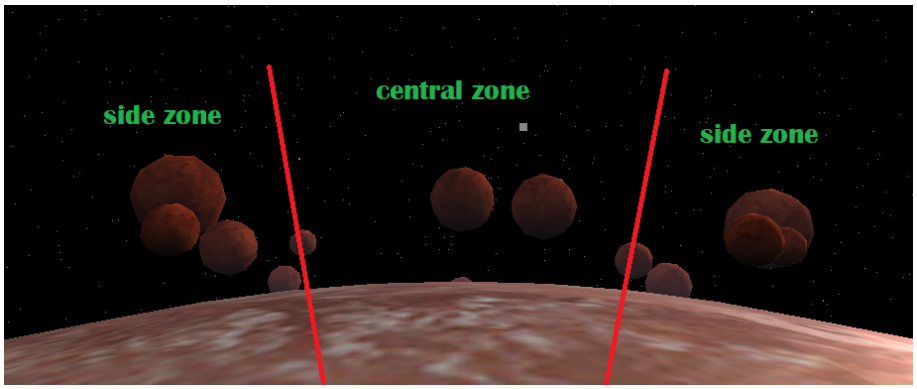
2.2.1 planets

The main planet is created in the *addWorld* function using *THREE.sphereGeometry* to form a sphere of given radius and *THREE.MeshStandardMaterial* to set color, shading and texture. For a more realistic effects it should be necessary to add also a "specular" and a "normal" version of the texture, but it was found that this produce a good result only from a far view of the model: for our case, with the camera so close to the texture, it's preferable to add only the specular one. At the end the world is added and positioned into the scene.



complete view of the main world

The rocks instead has been created as *THREE.IcosahedronGeometry* giving a random radius and a second parameters equals to 1 in order to shape them in a different way. They also keep the same texture of the main world but with different color and roughness. They are added as child nodes of the bigger planet, but their positions and orientation are randomized to perform a different environment each game. Actually there are 2 groups of rock: the real ones, animated and positioned in a central bound of the way that are involved in the game, and the externals, positioned over that bound, that are statics and serve only as scenery.

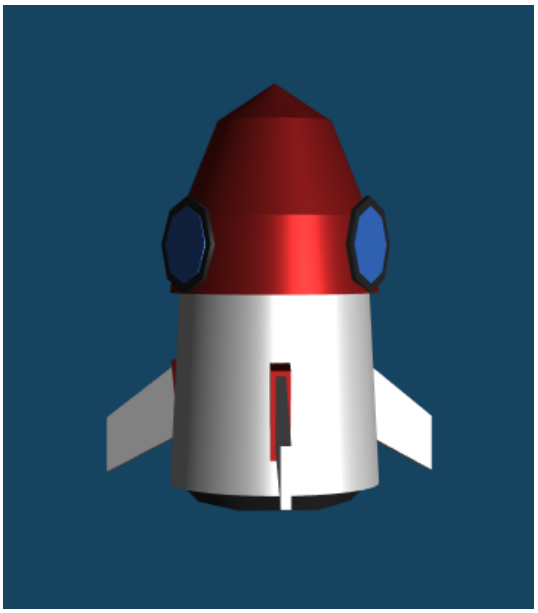


Different locations of rocks

2.2.2 space ship

The spaceship model is a bit more complicated than the planets one. The body it's formed by 5 consecutive *THREE.CylinderGeometry*, shaped as cones when needed. They are connected in a hierarchial way, starting from the top and staking each of the other part along the y axis. The window is also formed by 2 flat cylinder. For the wings instead has been used a dedicated function, that uses 2 *THREE.BoxGeometry* and shapes their vertices in the wanted form. Then 3 wing objects are connected at the spaceship "body" part, at the same distance each other. At the end of the rocket is placed also a last (animated) object to perform the fire from the engine, but this will be explained better in the particles section of this report.

For the materials has been used 3 textures (red metal, white metal and glass), and just a color for the bottom part.



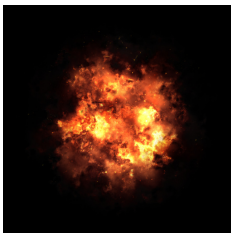
The Spaceship

As for the planets also the spaceship is added to the scene once it's created, then it's positioned just above the main planet and in front of the camera.

2.2.3 particles

In Three.js particle system works like any other primitive shape in that it has geometry and position, scale and rotation properties. What it does is use the individual vertices of the geometry to position the particles.

In this project they are been used to perform the explosion effect and the fire behind the spaceship. Both are created with a square geometry, then the correct shape is given overlaying them with a suitable sprite (or texture) and enabling the transparency.



Flame sprite

2.3 Animation and controls

All the models and the particles effects explained in the previous section are animated. This is made possible by the *update* function, that recursively call itself by *requestAnimationFrame(update)* and recalculates the parameters of the various model, for example the orientation of the planet and the rock generating rotation.

The movements of the spaceship are controlled by the user. To allow this is necessary to initialize some listener, two in our case: *document.onkeydown* and *document.onkeyup*. The first one reacts when one key is pressed or keep pressed, the second when the key is released. The involved keys and the relative effects are declared in the two functions. They are:

- Enter: works only once and make the game start setting a flag. It have effect only in press action;
- Directional arrows: While they are pressed the orientation of the spaceship changes so the rocket looks to the desired direction, then keeping pressed also its position (on x or y axis) is incremented or decremented until it reaches a determined bound. When the arrow is released the spaceship return to look forward and the movement is interrupted;
- Space Bar: When it's kept pressed it active the boost effect changing the value of a flag. So, if the variable representing the fuel is bigger than 0, the main planet increment its rotation speed and some game rules are changed (see User Manual section for more explanation). On the release of the bar the flag is changed to its initial value and the game returns to act normally.

The spaceship actually is free to move only in a vertical plane: the effect of advancing in the third dimension (and the different velocities) is given by the rotation of the world along the x axis in the counter direction.

The effects of the hierarchical modeling are particularly visible in two points: the first one is the floating effect of the rocks: they are actually firm, but the rotation of the world make them rotate with it. The second one is the rotation of the spaceship's wings: also in this case they are firm, their movement is given by the rotation of the part of the rocket where they are connected, while the rest of the model is firm too.

About the particles, as said before, it's possible to control the vertices in order to manage the position. So their animation process consist of updating, with random values, this vertices generating an free spreading effects of all the particles.

2.4 Game Logic

Game functioning is explained in the User Manual section, here are explained how the different features work and are implemented.

2.4.1 interacting with rocks

Rocks of the central section (see Create the Model) are created relying on a time interval: with the first update the clock starts, then when it reach the desired time the game execute a function that created a pool of rocks and collect them in a list. The clock is so restarted and the process repeated. Each time rocks are presents the game goes through the list and for each of them calculates the distance between it and the spaceship: if this distance is smaller than a determined value a collision is found. When collisions happen the rock involved is removed, the planet rotation speed (that simulates the rocket velocity) is slowed down, the explosion animation is activated generating the particles and the health value is reduced. The rocks not involved in collisions disappear (and are removed from the list) when they pass over the camera.

2.4.2 boost

Boost is a particular game feature: it's allows to increase the travel speed of the spaceship (that means more points in fewer time) and to receive lesser damage from collisions. As said before the boosting is activated only if there is still fuel. The fuel counter is set to 100 when the game starts, then it decrease continuously while the boost is on. When it reach 0 the game returns to works normally, so it's not possible to use it for long time periods. Once it's used the fuel is automatically restored: when the boost is not active the counter is increased, but this happen slower than the decreasing phase. The current state of the fuel can be seen in real time by a bar in the HUD.
During the boost also the particle effect of the fire is modified, making the geometries bigger to simulate an increase of power.

2.4.3 difficulty level

Before starting the game it's possible so select a level of difficulty. This choice don't modify the number of obstacles present in the scene, but instead reduce or increment the interval of time that determine the spawning of them. This affects the density of rocks presents during the game.

2.4.4 game over

The game ends when health goes to zero. In this case the spaceship is removed from the scene and the rotation of the main planet is stopped (the rotation of the rocks instead it's kept on for aesthetic reasons). It's also created and added the last component of the HUD that advise the player of the end of his run.

Chapter 3

Game User Manual

The scope of the game is to cover the longest distance possible with the spaceship avoiding the rocks floating around you. Every time you hit a rock your health is decreased, until the game over.
Use the directional arrow to control the spaceship and use the space bar to boost the rocket: during boosting the travel speed is increased and the damage taken is reduced, but be careful because it's not possible to boost when fuel it's empty.
On the lower left corner are reported the current values of scores, health and fuel.
Before starting the game it's possible to select between three difficulty levels, that change the number of obstacles present. If no choice is made the game will start with normal difficulty.

Chapter 4

Conclusions

This project was develop in order to apply, understand and test some of the important features provided by three.js. During its making it was possible to discover how this library simplify a lot of process, such as the creation of objects giving simple primitives and materials, realization of animation and so on.

The game created it's really simple and poor compared to the almost endless possibilities offered by three.js: with few additions for example it will be possible to create more planets and allow the spaceship to move from one to another, implement different types of obstacles or any other objects or add more action and animation (with relative effects) like shooting.