Interactive Graphics

Chichi Francesco, Jary Pomponi

June 27, 2017

1 List of all the libraries, tools and models used in the project but not developed by the team

- **Three.js:** Three.js is a cross-browser JavaScript library/API used to create and display animated 3D computer graphics in a web browser exploiting the power of WebGL in an high level mode.
- **OrbitControls.js:**
- **Detector.js:**
- **stats.min.js:**

2 Description of all the technical aspects of the project

2.1 Ship

- **Glass:**
- **Cabin:**
- **Ring.js:**
- **Motors.js:**

2.2 Halo

The halo object is used to simulate the day-night cycle and it is implemented in the Halo.js file.

We have implemented the halo object with a group, which is composed by a torus and two spotlight, one for the sun and the other for the moon. Those spotlight are positioned in opposite part of the torus and both points to the centre of the game plane. The difference between the lights is the colour of the light. In order to simulate the cycle we used a rander function, that will rotate the torus by 0.5 degrees each frame.

The modality of the Halo can be choose in the main menu; we have:

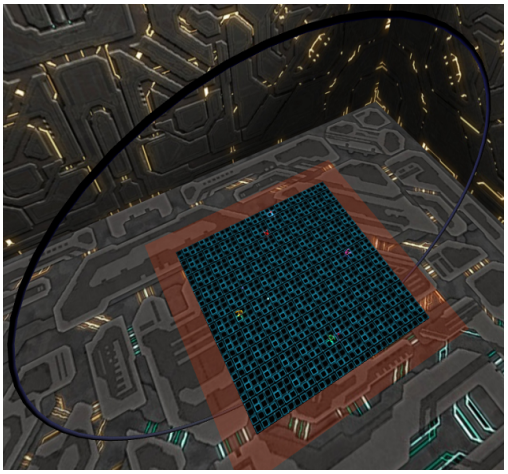

Figure 1: Day modality

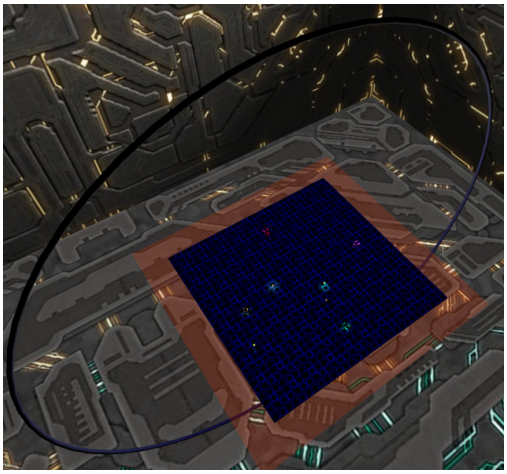

Figure 2: Night modality

- day: the torus starts in the day configuration and won't rotate.
- night: the torus starts in the day configuration and won't rotate.
- cycle: day: the torus starts in the day configuration and will rotate.

The differences between day and night can be seen in the figure above.

2.3 Animated Light

An animated Light is a sphere that emits light and move in the space. This is implemented in the animatedLight.js file.

Each light is a point light, and is constructed by generating a set of random points in the space, delimited by the plane space, and a random colour. Then those points are interpolated using a closed loop CatmullRomCurve3, from THREE. Since the position of the light in the curve is defined in the interval from 0 and 1, we increment the position, at each frame, with a small delta, in order to simulate the movement . This is done is the render function.

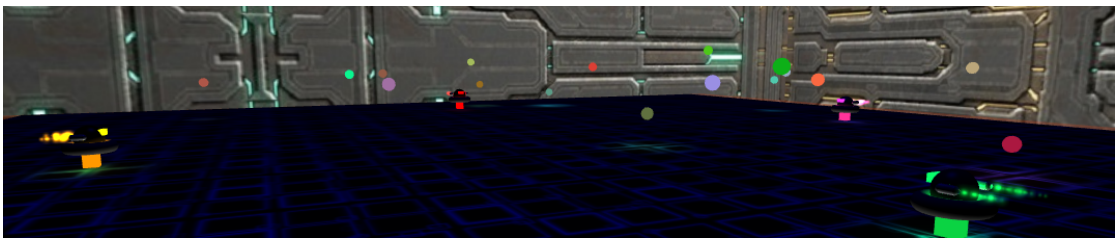

Figure 3: some animated lights

2.4 Plane and skybox

The plane is a transparent square, in which the ships can move; the material is a Phong type. Below the plane we have a red transparent flat square, that indicates the zone in which the ships cannot go. The material of the red square is a basic mash material.

The skybox is simply the composition of two texture, one repeated two time, for the bottom and the top, and another repeated and rotated four time. Those texture composed form the skybox cube.

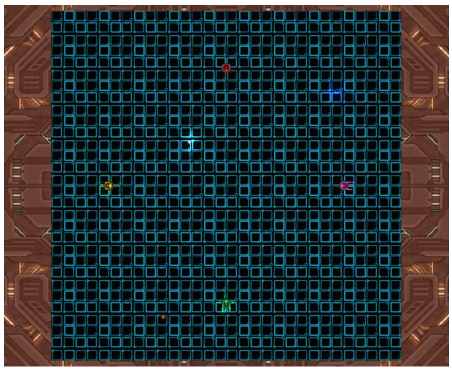

Figure 4: the plane and the red square

3 Game Logic and rendering

In this section we will analyse the game logic, the render part of the game and the animations of the ship.

3.1 Main loop and render

The main loop is the *animate* function.

If the game is running we check for collisions and see if the game ending conditions are met. Respectively using the *collision* function and the *checkEnd*. The collision logic will be explained later. In the *checkEnd* we also move the players; the movement of the player will be explained in *animations and player movement*.

If the game is on pause we check for every button is pressed, and based on the state of the game we display some menu; the menu section will cover this part. Then, depending on the active scene, the game or the menu, the function will call a different render function.

In the render of the menu we rotate the ships, update theirs particle motors and modify the colour of each ship, depending on the chosen one, while in the render of the game we move the ships, the animated lights and rotate the halo, if the cycle is selected as modality of the light. Moreover we set the camera to follow the player if the single player mode is selected. In the render of the game we also animate the ships; the animation *animations and player movement*.

3.1.1 Collision

The collision are implemented using the Box3 class from Three.js. For each ship we compute te bounding box of the cabin, and for each wall we compute the bounding box of the entire wall segment. Each player control contains the box of the cabin and an array with the box of the players walls. The collision check consist of some basic controls inside some loops.

If the player cross the plane boundary then will die. Elsewhere, for each player, we iterate over all the walls and over all others player's ship box. If the collision test with walls will result positive then the player will die, but if the player collides with another player both will die. If a player is dead there is no reason to check collision.

3.2 animations and player movement

Each ship is animated in the render function, calling the render function in the player.js file. The animation consists in moving the ship on y axis following a sin function and rotating the torus by certain amount of degrees. Moreover all the particles of the motors will be updated.

In this file we check if the player is dead or not. If it is dead then the explosion animation will start. The explosion animation consist on a certain amount of particles

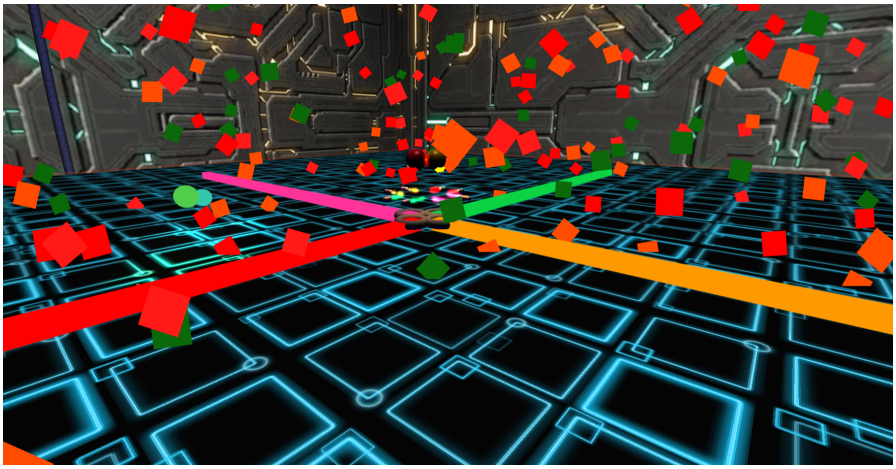

Figure 5: Collision between 4 ships

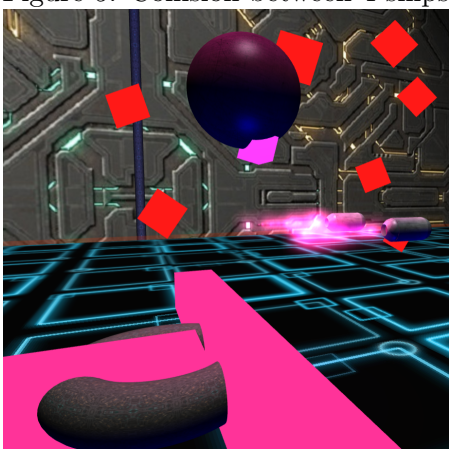

Figure 6: Animation of the explosion

(sprites) and an animation of the ship.Each particle will move from the explosion point to a random point in the plane, with a parabolic movement, using CatmullRomCurve3 from *Three.js*. Like in the case of the animated lights, the curve position is defined from 0 to 1. So each frame start at 0 position, and when reach the position 1 will be deleted. In the animation of the ship the motors will continues to go straight ahead, the torus will fall on the ground and the cabin with the glass will rotate and going up. Meanwhile the ship and the will shrink.

When the animation is over the ship and the walls will disappear from the scene.

3.3 Menu

3.3.1 Kyeboard settings

3.3.2 Color settings

4 Description of the implemented interactions