SAPIENZA
UNIVERSITÀ DI ROMA

Master of Science in Engineering in Computer Science

A.Y. 2018-2019

Final project Interactive Graphics:

Ianna's plane

***Student**:*

Marco Magnani 1532000

***Professor**:*

Marco Schaerf

Index:

Short introduction on ThreeJS:

Three.js is a cross-browser JavaScript library and Application Programming Interface (API) used to create and display animated 3D computer graphics in a web browser. Three.js uses WebGL. Three.js allows the creation of Graphical Processing Unit (GPU)-accelerated 3D animations using the JavaScript language as part of a website without relying on proprietary browser plugins. This is possible due to the advent of WebGL.

This library is widely used as it provides many constructs as we will see in this project.

Ianna's plane project:

Introduction about project:

This project consists of a game. The game in question belongs to the category of endless games, that is a type of game that never has an end if not for direct loss of the player.

The game was called Ianna's plane, as the user has control of a small airplane which will have to dodge objects to get a score.

Ianna's game is a game developed with the use of ThreeJS.

At the start of the game the user can choose the difficulty of the game. The user can choose between: easy, normal and difficult. The 3 modes differ depending on the number of bullets that will be created in games.

Structure of the project:

The structure of the project is very basic and well-ordered. Below is a brief explanation of the files and directories.

Directories:

- *lib*: list of libraries for the project.
- *css*: contains css file for the project
- *img*: contains image for the project
- *sound*: contains audio file for the project

Files on root:

- *index.html*, this is the first file that the user will see when you open the game. Here the user can choose the mode.
- *points.html*, this file is the last view of the game, when the user finishes the game, he is redirect to this page.
- *iannaPlane.html*, this file is the html heart of the game.
- *iannaPlane.js*, this file is a JavaScript file and it is the heart of the game, because contains all logic part.

In this project, I am used some components of ThreeJS: Stats and *orbitControl*. These files are in *libs* folder.

The code:

Flow:

As we have already said, the user must follow a file flow. The first file the user encounters is *index*.*html*. This file shows the commands to play and the user has the possibility to choose the difficulty of the game: easy, normal, difficult. By default it is set to easy. As soon as the user presses the button for the game he will be taken to *iannaPlane*.*html* where the real heart of the game is and where logic is present. When the user ends the game because he loses, he will be asked whether to play again, and a refresh of that file will be done otherwise he will be taken to the last page *points*.*html*, where he will be thanked for the game.

Models:

Inside the game, there are several hierarchical models: trees, bullets and airplane.

Airplane:

The airplane is modelled with the *Airplane* object inside the code and then added to the scene with the *createPlane*() function.

Speaking of the Airplane object, a model of the format with various parts is formed: pilot, cabin, engine, left and right flap, two tails, wings and propeller.

For each part a shape is defined with the function of ThreeJS *BoxGeometry* which allows to draw a box of certain dimensions.

The important function that is used for each part is the ThreeJS *MeshPhongMaterial* function, which allows you to define the properties of the material. I chose the Phong model because, this is much better than doing it in the application, because it could be overloaded by the CPU by sending too much data to the GPU.

Next, for each part, properties are set for the shadows and the position of the object.

```
// Create the pilot
var geometryPilot    = new THREE.BoxGeometry(0.5,0.2,0.5,1,1,1);
var materialPilot    = new THREE.MeshPhongMaterial({color:'white', shading:THREE.FlatShading});
var pilot            = new THREE.Mesh(geometryPilot, materialPilot);
pilot.position.set(0,0.6,0);
pilot.castShadow        = true;
pilot.receiveShadow     = true;
this.mesh.add(pilot);
```

Here in the picture we can see the modelling of the pilot.

At this point we have created the airplane object and we just have to add it to the scene, we can do this with the *createPlane*() function.

In this function the position of the entire model on the plane is set and the lane is set.

The airplane is equipped with movements, that is the rotation of the propeller and the movement of the flaps when you change position, but the latter will be seen later.

Tree:

The second important model of the game is the tree. The tree also is a hierarchical model composed of several parts: a trunk and leaves.

The leaves part was made using the ThreeJS *ConeGeometry* function, which gives me the shape of a cone. The part of the trunk was made with the ThreeJS *CylinderGeometry* function, which gives me the shape of a cylinder.

These two parts were put together with object modeling.

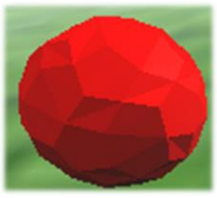
As for the airplane, the Phog model for material treatment was set.

```
is function is used to create to tree. This is hierarchic model: trunk and top
nction createTree(){
    var sides = 8;
    var tiers = 6;
    // Set the cone geometry and the material
    var treeGeometry = new THREE.ConeGeometry( 0.5, 1, sides, tiers);
    var treeMaterial = new THREE.MeshPhongMaterial({
                                                        color: 0x33ff33,shading:THREE.FlatShading
                                                    });
    var treeTop = new THREE.Mesh( treeGeometry, treeMaterial );
    // Set some properties: shadown and position
    treeTop.castShadow      = true;
    treeTop.receiveShadow   = false;
    treeTop.position.y      = 0.9;
    treeTop.rotation.y      = (Math.random()*(Math.PI));

    // Set the cylinder geometry and the material
    var treeTrunkGeometry = new THREE.CylinderGeometry( 0.1, 0.1,0.5);
    var trunkMaterial = new THREE.MeshPhongMaterial( { color: 0x886633,shading:THREE.FlatShading  } );
    var treeTrunk = new THREE.Mesh( treeTrunkGeometry, trunkMaterial );
    // Set some property position
    treeTrunk.position.y = 0.25;

    // I put together the the parts
    var tree =new THREE.Object3D();
    tree.add(treeTrunk);
    tree.add(treeTop);
    return tree;
}
```

Then the properties for shadows and position are added.

A small graphical effect given to the tree to seem more natural is the rotation property. That is, the tree is tilted a small random number so that it has a more natural effect.

Bullet:



The other model in the video game is the bullet. The bullet is the real obstacle that the user has and if he hits it he loses the game.

As with the other models, the bullets are also created with a ThreeJS *DodecahedronGeometry* function with one radius, then the material is added with the Phong model.

```
// This function is use to create the red bullet model
function createBullet(){
    // To make the bullet I used a dodecahedron geometries. Here, I set the material,geometry and position.
    var bulletGeometry = new THREE.DodecahedronGeometry( bullet_radius, 1);
    var bulletMaterial = new THREE.MeshPhongMaterial({
                                                        color: 'red' ,shading:THREE.FlatShading
                                                    });
    var bulletTop           = new THREE.Mesh( bulletGeometry, bulletMaterial );
    var bullet              = new THREE.Object3D();
    bulletTop.position.y    =   0.9;
    // Finally, I add the bullet
    bullet.add(bulletTop);
    return bullet;
}
```

Then the property for the position on the y-axis is set.

Environment:

The environment is all set in the *createScene*() function as all the main functions are called here.

In this function the dimensions for canvas and the scene are set. The scene is the heart of all the architecture behind the game. Here the position of the sound, camera, the texture for the background and the events for both *onResize* and *onKeyDown* are set.

On the *onResize* event is set the function for the resize of the canvas to have a very strong user experience while on the *onKeyDown* event is set the function for the movement of the in-game airplane.

Texture:

The world that rotates is created with the *createWorld*() function. In it a sphere is created and the material is combined. In the function of the material the texture is loaded from an image. The picture is about a lawn.

```
// This function is used to create the world
function createWorld(){
    var sides = 40;
    var tiers = 40;
    // Load a texture from one image.
    var texture = new THREE.TextureLoader().load('img/prato.png');

    // I make the Earth with geometry and material
    var earthGeometry    = new THREE.SphereGeometry(worldRadius,sides,tiers);
    var earthMaterial    = new THREE.MeshBasicMaterial({
                                                        map: texture
                                                    });
    rollingEarth = new THREE.Mesh( earthGeometry, earthMaterial );

    // I set the properties of the Earth
    rollingEarth.receiveShadow      = true;
    rollingEarth.castShadow         = false;
    rollingEarth.rotation.z         = -Math.PI/2;

    // I add the Earth to scena.
    scene.add(rollingEarth);
    rollingEarth.position.y         = -24;
    rollingEarth.position.z         = 2;

    // I call the tree function.
    createSetTrees();
}
```

As for the models, it is also possible to see it as both material and form are put together in a mesh.

The properties for the shadows and the rotation on z of the earth are set.

The second texture that we find in the game is the background of a sky. It is also loaded with an image and placed in the background.
This is done in the *createScene*() function.

```
// Set the texture for the sky
var texture_sky      = new THREE.TextureLoader().load('img/cielo.jpg');
scene.background     = texture_sky;
```

Light:

The light is added through a call to *createLight*() function in the *createScene*() function.
In the function *createLight*() the light is added with the use of the ThreeJS *HemisphereLight* function and always positioned with the use of the *DirectionalLight* function belonging to ThreeJS.

```
// This function is used to add and create the light for the world
function createLight(){
    // I make the light and I set the color.
    var earthLight = new THREE.HemisphereLight(0xfffafa,0x000000, .9)
    scene.add(earthLight);

    // I set a direction of the light.
    sun = new THREE.DirectionalLight( 0xcdc1c5, 0.9);
    sun.position.set(12,6,-7);
    sun.castShadow = true;
    scene.add(sun);

    //Set shadow properties for the sun light
    sun.shadow.mapSize.width    = 256;
    sun.shadow.mapSize.height   = 256;
    sun.shadow.camera.near      = 0.5;
    sun.shadow.camera.far       = 30 ;
}
```

Here the properties of light such as color and shadows are set.

Sound:

The sounds in the game are 2: one for the background that recalls the sound of the planes of the airplane that turns and the other is the sound of an explosion that only activates when the user hits a bullet.

The sounds are set in the *setAudio*() function which creates 2 listeners are the components added to the scene that give the task of reproducing the audio loaded into the *Audio* objects.

```
unction to set the audio
unction setAudio(){
    // I create the 2 listener and i add them
    var listener     = new THREE.AudioListener();
    var listener_2   = new THREE.AudioListener();
    camera.add(listener);
    camera.add(listener_2);

    // I create the 2 sound and I link with listener
    sound_background    = new THREE.Audio(listener);
    sound_explosion             = new THREE.Audio(listener_2);

    // load a sound and set it as the Audio object's buffer
    var audioLoader = new THREE.AudioLoader();
    audioLoader.load( 'sound/propeller.mp3', function( buffer ) {
        sound_background.setBuffer( buffer );
        sound_background.setLoop( true );
        sound_background.setVolume( 0.5 );
        sound_background.play();
    });
    audioLoader_explosion = new THREE.AudioLoader();
    audioLoader_explosion.load( 'sound/explosion.mp3', function( buffer ) {
        sound_explosion.setBuffer( buffer );
        sound_explosion.setVolume( 0.5 );
    });
}
```

Audio files can be found in the *sound* folder. The background audio is loaded and immediately started via the play function, while the explosion audio is simply loaded and placed at the scene, and at the time of the explosion it will be started.

Logic:

In this chapter we will talk about the logic of Ianna's plane, that is, how objects are added to the world, how they are created and how the user can interact with the game.

Let's start by talking about how trees are added to the world.

In the *createWorld*() function a call is made to *createSetTrees*(). This function takes care of generating a set of trees for a maximum of 72 alternately.

Speaking of the bullets they are generated by continuation in the update function.

The *update* function is called because it deals with the updating of objects, in fact inside it we find: the calculation of the score, the control if the user

has hit a bullet, the regeneration time of trees, propeller animation and world rotation animation.

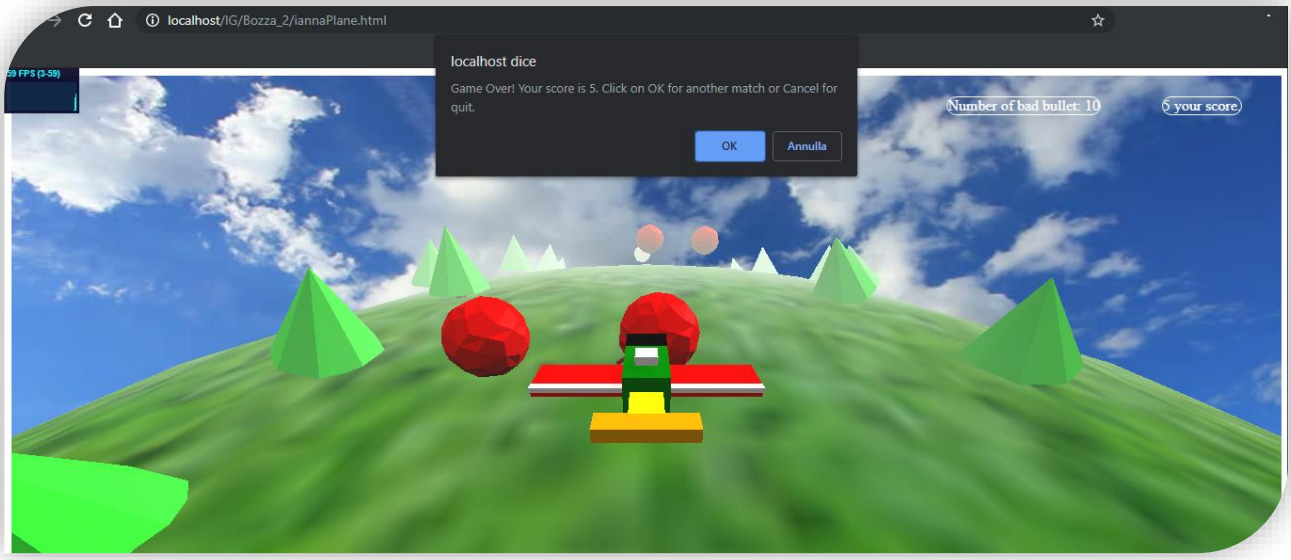
Now we'll talk about how the question of when a bullet is hit is handled by the game.

```
his function is used to set the logic bullet, that is when I hit or dodge a bullet
nction bulletLogic(){
    var oneBullet;
    var bulletPos       = new THREE.Vector3();
    var bulletsToRemove = [];
    bulletInPath.forEach( function ( element, index ) {
        oneBullet = bulletInPath[index];
        bulletPos.setFromMatrixPosition(oneBullet.matrixWorld);
        if(bulletPos.z > 6 && oneBullet.visible){           //gone out of our view zone
            bulletsToRemove.push(oneBullet);
        }else{                                  //check collision
            if(bulletPos.distanceTo(airplane.mesh.position) <= 1.2){    // If I am here, I hit a bullet
                sound_background.stop();                            // Stop the background audio
                sound_explosion.play();                             // Lunch the explosion audio
                youLose = true;
                // I ask to user if you play again
                var confirm_check = confirm("Game Over! Your score is "+score+". Click on OK for another match or Cancel for quit.");
                if (confirm_check == true) {        // Yes, I reset the game
                    location.reload();
                    cancelAnimationFrame( globalRenderID );
                    window.clearInterval( powerupSpawnIntervalID );
                } else {                    /// Redirect him on another page.
                    window.location.href = "points.html";
                    cancelAnimationFrame( globalRenderID );
                    window.clearInterval( powerupSpawnIntervalID )
                }
            }
        }
    });
```

The logic of the bullet can be found in *bulletLogic*(). In this function the distance between the airplane and the nearest bullet is continuously calculated, if it is less than 1.2 (value chosen by trial and error based on game tests), it is assumed that the user is too close to the object and so we can conclude that he hit the bullet.

When this situation occurs, a set of procedures is started:

- background audio is stopped
- the explosion sound is started
- *youLose* is set to true, that is it indicates that the game must end and therefore the calculation of the points must stop.
- a confirmation popup is started, which in addition to showing the user his score, he is asked if he wants to make another game. In case you choose to play again, the game is reset otherwise the user has chosen not to want to do another game will be brought to the last screen above we talked about.



User interaction:

At this point we have to talk about how the user can interact with the airplane.

As we have already said the user can pilot the plane with the keyboard arrows: right and left. The use of arrows is essential to avoid bullets.

As we have seen, in the *createScene*() function the *handleKeyDown*() function is set on the *onkeydown* event.

This function listens on pressing the keyboard keys.

The *handleKeyDown*() function is sufficient on recognizing the pressure of the key 37 that is left arrow and 39 that is right arrow.

We will only talk about the left arrow, but the same is true for the right arrow.

When the button is pressed controls are made to see if the next position is valid or not, for example if we are on the left side and the user presses left nothing must happen as we are on the edge and therefore it is an invalid move.

```
unction is used from the user to move the airplane.
unction handleKeyDown(keyEvent){
    var validMove = true;       // I use this variable to check if the moving is valid.
    // If the user presses on left arrow of keyboard the airplane will move on the left side if the moving is valid.
    if ( keyEvent.keyCode === 37) {         //left moving
        if(current_track == center_track){  // I check if I am on the center track.
            current_track                   = left_track;
            airplane.leftFlap.rotation.x    = 10;   // Move the flap
            airplane.rightFlap.rotation.x   = -10;
        }else if(current_track == right_track){ // I check if I am on the right track
            current_track                   = center_track;
            airplane.leftFlap.rotation.x    = 0;
            airplane.rightFlap.rotation.x   = 0;
        }else{                                  // Otherwise I am on the left side, bad move
            validMove = false;
        }
    } else if ( keyEvent.keyCode === 39) { //right moving
        if(current_track == center_track){  // I check if I am on the center track.
            current_track                   = right_track;
            airplane.leftFlap.rotation.x    = -10;  // Move the flap
            airplane.rightFlap.rotation.x   = +10;
        }else if(current_track == left_track){  // I check if I am on the left track
            current_track                   = center_track;
            airplane.leftFlap.rotation.x    = 0;    // Move the flap
            airplane.rightFlap.rotation.x   = 0;
        }else{                                  // Otherwise I am on the right side, bad move
            validMove = false;
        }
    }
    airplane.mesh.position.x = current_track;
}
```

If the move is valid, the airplane must move to the desired lane.

A small graphic effect created on this project is the movement of the flaps, that is directional wings that the planes have in the real world when I want to change course.

Depending on the movement desired by the user, you will see a movement of the right and left flap.

On Resize:

We talked about the resize window event. On the event *window resize* is in listen a function *onWindowResize*() that changes the size of the game based on the size of the window that the user has available. This feature has been implemented to improve the user experience

```
// This function is responsible to set the auto resize the canvas dimension.
function onWindowResize() {
    canvas_width    = document.getElementById('canvas').offsetWidth;
    canvas_height   = document.getElementById('canvas').offsetHeight;
    renderer.setSize(canvas_width, canvas_height);
    camera.aspect = canvas_width/canvas_height;
    camera.updateProjectionMatrix();
}
```

Mode:

Finally, we will talk about the choice of mode.

As already mentioned at the beginning of the game the user is sent to choose a game mode: easy, normal or hard.

The three game modes differ in the number of bullets that the user must avoid.

The bullets for the 3 modes are:

- easy -> 10 bullets
- normal -> 20 bullets
- hard -> 30 bullets

When the user chooses the mode from the *index*.html page, a cookie is set in the browser that saves the value chosen by the user.

This value is retrieved from *iannaPlane*.js with the *takeCookie*() function.

This function retrieves the cookie value and returns it as a value.

If the user has not enabled cookies on his browser or uses an extension to block them, the easy mode will be chosen as default.

```
// Function is use to take value from cookie, that is the type of mode choosen from user.
function takeCookie(){
    // I take the name
    var name = "ModeIannaPlane=";
    var decodedCookie = decodeURIComponent(document.cookie);
    var ca = decodedCookie.split(';');
    for(var i = 0; i <ca.length; i++) {
        var c = ca[i];
        while (c.charAt(0) == ' ') {
            c = c.substring(1);
        }
        if (c.indexOf(name) == 0) {     // When I find my cookie, I return its value.
            return c.substring(name.length, c.length);
        }
    }
    // if i return here, the user did not choose the mode or he does not allow the cookie use
    // I set the easy mode the default
    return 10;
}
```