Interactive Graphics

Guardian of the Galaxy

Federico Bucci 1649197

Gabriele Sellani 1709058

Michael Capponi 1711759

Riccardo Fiore 1705634

Contents

1 Introduction

The project chosen to satisfy all requirements is a survival video game which is called "Guardian of the Galaxy".
The user is immersed in a space environment driving a starship to counteract the enemies of the galaxy.
The aim of the player is to survive as much as possible against the hordes of enemies. The game is structured in a finished set of levels of increasing difficulty that are repeated over time.
Whenever the user wins three levels the number of enemies remains always the same, but their speed and their attack power increases more and more.

During the game, coins are generated and they are used by the player to buy upgrades during the game.
The upgrades available are:

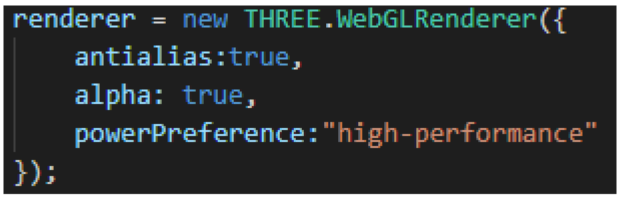
1. Restore health
2. Increase damage

The user can control the movement of the starship with simple commands:

- The movement in the space of the starship is managed with WASD keys (W and S to move the starship on the vertical axis, while A and D to move the starship on the horizontal axis).
- The orientation of the cursor that regulates the shot of the starship is managed through the movement of the mouse.
- Each click of the mouse corresponds to a shot.
- The E and P keys are used to start the game by entering the portal and returning to the home page.
- The R key is used to restart the game without reloading the models.

2 Three.js: environment and scene

Three.js is a JavaScript framework for rendering interactive 2D and 3D graphics on the web, providing the possibility to display incredible models, texture, music and videos, data visualizations.
So, the aim of Three.js is to create an easy to use and lightweight 3D library. It also provides WebGL renderers, to support 3D animations.
From a snippet of our code we can see our renderer:



```
renderer = new THREE.WebGLRenderer({
    antialias:true,
    alpha: true,
    powerPreference:"high-performance"
});
```

Figure 1: Renderer

In the WebGL renderer we set the parameter anti-alias at true so that we can reduce jaggies (something like lines that should be smooth but they are not due to the not enough resolution of the device monitor) by surrounding them with gray-scaling. Then we set alpha parameter at true to support transparency in order to set our background and finally we set high performances to support rendering.
We can define a graph to explain how Three.js environment works. We have seen that exists a renderer, below it we have the **scene**.
The scene is an object that allows to set up all those things we want to be rendered, so for example objects' models, lights and camera.
About the lights we added two:

- An **AmbientLight** that globally illuminates each object of the scene equally. We set its color white as default and an intensity of 0.5
- A **PointLight**, white, to illuminate each direction from a single point to strongly illuminate objects from a specific point of view. We put it in the left of the scene.

Then we defined the camera, that is a main point for the game: it sets the point of view from which the players see objects and play. For this reason, as we want to build a third-person view, we put the camera behind the position we will set our starship: we defined a **PerspectiveCamera** that is a camera that uses a perspective projection in order to reproduce a realistic way of how human eye sees and so to reproduce a 3D view. The first two parameters for the perspective camera are **fovy** and **aspect**: the fovy parameter which stands for "field of view y-axis", is the vertical angle of the camera's lens and so it is expressed in degrees, 60 for our camera; the aspect parameter is simply a ratio parameter and it is the division of the window width for the window height.
The last two parameters, near and far are just values that defines which z values one want to be displayed by defining two "clipping planes".
Finally we have to talk about objects and models in the scene: these objects are classified as **Mesh** objects. Mesh class represents triangular polygon mesh based objects, that are a collection of vertices, edges and faces that defines a 3D model.

A Mesh object takes as input two parameters: geometry and materials objects.

- **Geometry** is a representation of mesh, lines and points that includes vertices, faces, normals and colors.
- **Material** is the appearence of the related Mesh, so it defines how to draw the object,basic, shiny, flat, what color, what texture (images, preocedural, specular ...) to apply etc. There will be different materials used in the project as many geometry will be defined, so we explain them in a more specific way later when talk about each object.

Here we can see the structure of a Three.js environment as a graph:

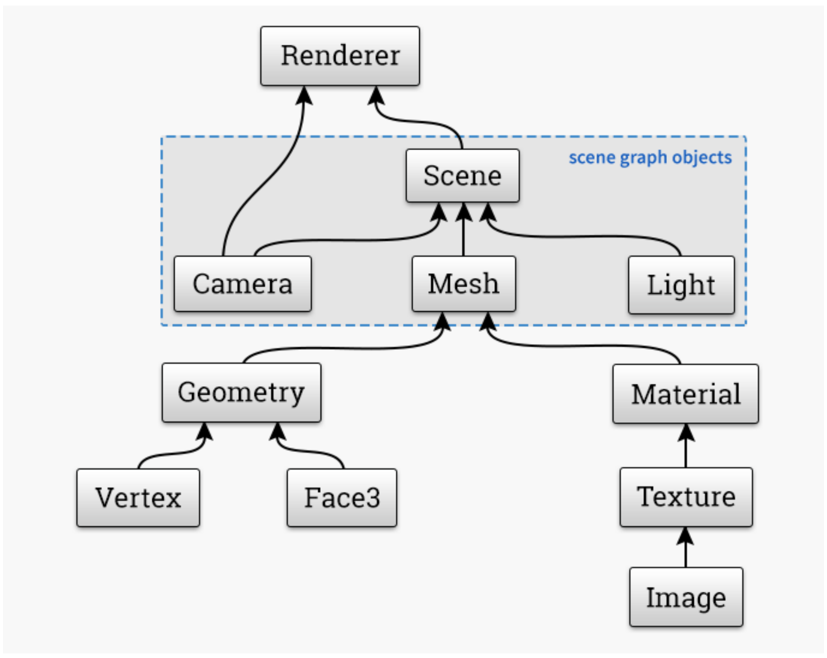


Figure 2: Scene Graph

3 Background scene - Space

In this section we will analyze the animated background. It is composed of graphic elements that give rise to a space environment.
The scenography was created using css elements and html elements and it is composed of two main elements: stars and twinkling.
The overlapping of the two divs (containers for other elements) inserted in the html file, together with the code inside the css file allows us to obtain an excellent final result.
The stars were created to define the background that is initially static: a black background and an image imported from the internet that represents a starry sky. The fundamental part is the twinkling which consists of a linear and infinite animation of the position of the background.
At the beginning we add also clouds which work the same as the twinkling but in the end, we decided to take them off, to avoid overloading the game too much.

We also worked on parameters to make the intensity of the twinkling more suitable for our game and not to cover up the other elements.

Figure 3: Snapshot of the animated space background

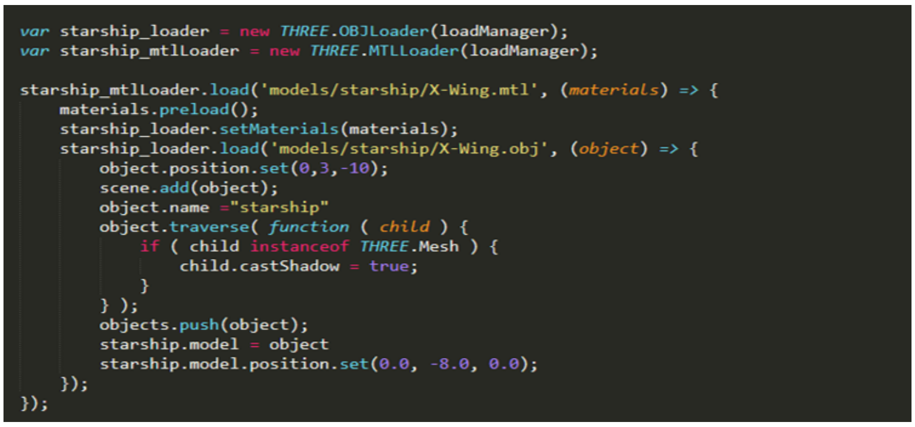
4 Models

The models represent one of the main parts of our project. Among the loaders that the Three.js library makes available we have extensively used the OBJLoader that is is a simple data-format that represents 3D geometry. Every model is represented on the 3 axes, below our models:

- ***STARSHIP*** It is the protagonist of our project. First of all we declared a "starship" variable to give it a structure for the purposes of the game logic.

```
starship = {
        model: null,
        name : "starship",
        money: 0,
        health: 10,
        damage: 1
};
```

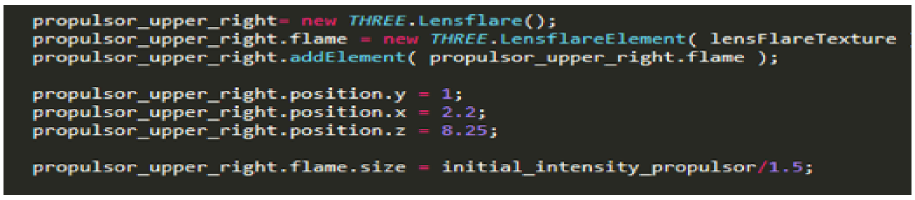
We imported a .obj hierarchical model found online, to use this we need a MTL-Loader which loads an .mtl resource, used internally by OBJLoader. A MTL File Format is a companion file format to .obj that describes surface shading (material) properties of objects within one or more .obj files.
In Three.js to handle and keep track of loaded and pending data let's use THREE.LoadingManager(). In this case, in the *loadModels(loadManager)* function we have:



```
var starship_loader = new THREE.OBJLoader(loadManager);
var starship_mtlLoader = new THREE.MTLLoader(loadManager);

starship_mtlLoader.load('models/starship/X-Wing.mtl', (materials) => {
    materials.preload();
    starship_loader.setMaterials(materials);
    starship_loader.load('models/starship/X-Wing.obj', (object) => {
        object.position.set(0,3,-10);
        scene.add(object);
        object.name ="starship"
        object.traverse( function ( child ) {
            if ( child instanceof THREE.Mesh ) {
                child.castShadow = true;
            }
        } );
        objects.push(object);
        starship.model = object
        starship.model.position.set(0.0, -8.0, 0.0);
    });
});
```

Through the function shown above we also gave it a name and a position within the scene.

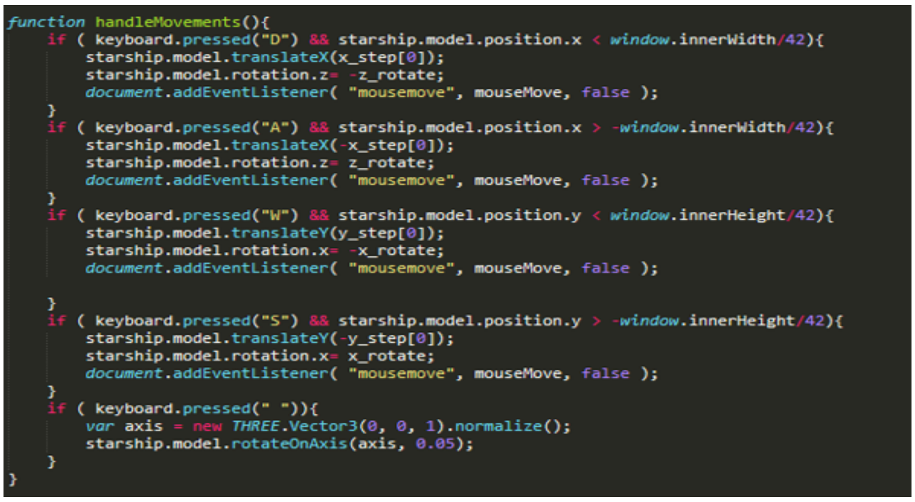
To make our model even more realistic we decided to insert the light generated by the 4 thrusters of the starship. We used the LensFlare.js library which creates a simulated lens flare that tracks a light.Once the texture has been defined and loaded with the TextureLoader, we have defined 4 new Three.Lensflare objects in the previous function. For example:



```
propulsor_upper_right= new THREE.Lensflare();
propulsor_upper_right.flame = new THREE.LensflareElement( lensFlareTexture
propulsor_upper_right.addElement( propulsor_upper_right.flame );

propulsor_upper_right.position.y = 1;
propulsor_upper_right.position.x = 2.2;
propulsor_upper_right.position.z = 8.25;

propulsor_upper_right.flame.size = initial_intensity_propulsor/1.5;
```

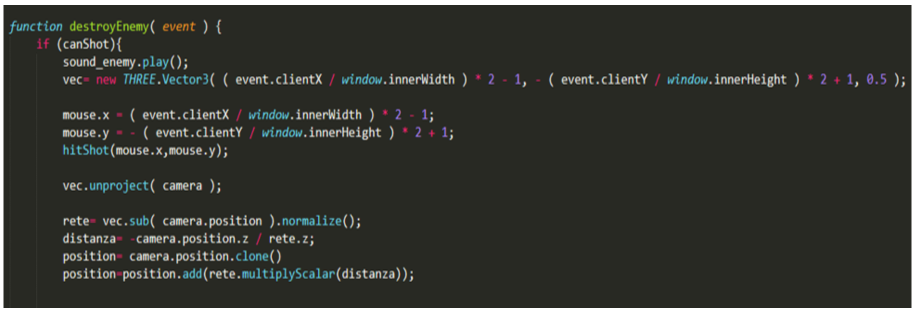
The same logic applies to the remaining thrusters where the position changes along the axes. Then the 4 thrusters are added when the starship is started by pressing the ”E” key at the beginning of the game (*playGame()* function)

To give an animation to our model we used the THREEx.KeyboardState.js library. We used it in the ***handleMovements()*** function (obviously called in the animate rendering function as all other animation handlers) where we have defined the keys corresponding to certain movements, how much the spaceship must translate and rotate along the axes. We have also added a control so that the ship does not come out of the camera (user’s screen). Finally, another automatic animation we added for the starship is that of open/close its wing when starting/ending a battle.



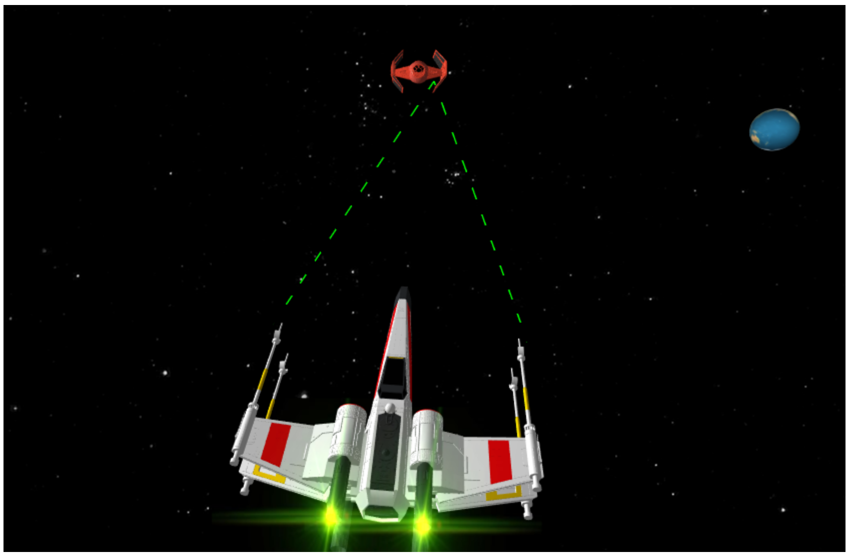
```
function handleMovements(){
    if ( keyboard.pressed("D") && starship.model.position.x < window.innerWidth/42){
        starship.model.translateX(x_step[0]);
        starship.model.rotation.z= -z_rotate;
        document.addEventListener( "mousemove", mouseMove, false );
    }
    if ( keyboard.pressed("A") && starship.model.position.x > -window.innerWidth/42){
        starship.model.translateX(-x_step[0]);
        starship.model.rotation.z= z_rotate;
        document.addEventListener( "mousemove", mouseMove, false );
    }
    if ( keyboard.pressed("W") && starship.model.position.y < window.innerHeight/42){
        starship.model.translateY(y_step[0]);
        starship.model.rotation.x= -x_rotate;
        document.addEventListener( "mousemove", mouseMove, false );

    }
    if ( keyboard.pressed("S") && starship.model.position.y > -window.innerHeight/42){
        starship.model.translateY(-y_step[0]);
        starship.model.rotation.x= x_rotate;
        document.addEventListener( "mousemove", mouseMove, false );
    }
    if ( keyboard.pressed(" ")){
        var axis = new THREE.Vector3(0, 0, 1).normalize();
        starship.model.rotateOnAxis(axis, 0.05);
    }
}
```

For the purpose of our theme we have implemented the shot against enemies and planets. In both cases we used the ***destroyEnemy()*** function,when an enemy appears the starship can rotate trough the mouse position (starship follows the direction of the mouse and rotates accordingly), this is a simulation of a fight between the star ships. Below there is a snippet of this code:

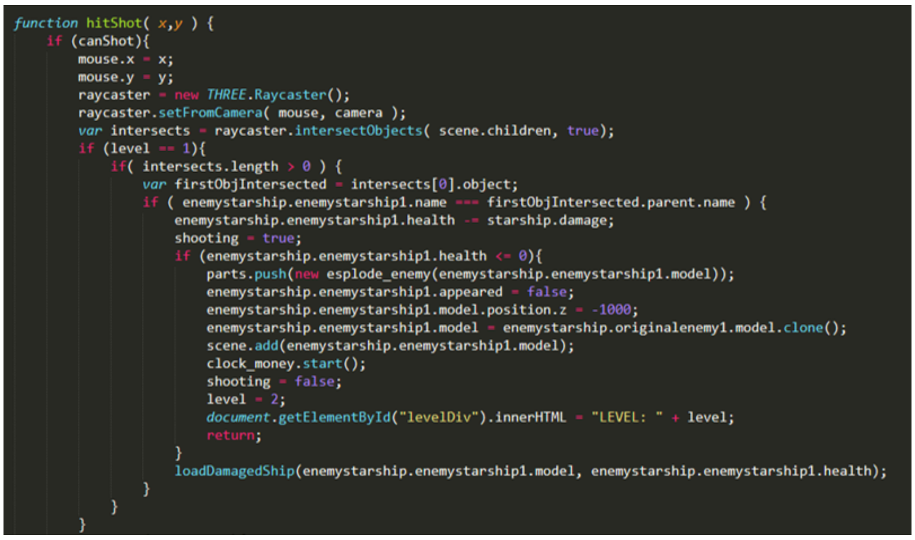


```
function destroyEnemy( event ) {
    if (canShot){
        sound_enemy.play();
        vec= new THREE.Vector3( ( event.clientX / window.innerWidth ) * 2 - 1, - ( event.clientY / window.innerHeight ) * 2 + 1, 0.5 );

        mouse.x = ( event.clientX / window.innerWidth ) * 2 - 1;
        mouse.y = - ( event.clientY / window.innerHeight ) * 2 + 1;
        hitShot(mouse.x,mouse.y);

        vec.unproject( camera );

        rete= vec.sub( camera.position ).normalize();
        distanza= -camera.position.z / rete.z;
        position= camera.position.clone()
        position=position.add(rete.multiplyScalar(distanza));
```

This function is always called in the listener's *mousedown*, the variable "canShot" is setted true when the battle starts, then the starship can shoot the enemy starships
In this function we considered the mouse position and it is created a direction from the starship to the mouse position. Then it is created a geometry object that represent the laser bullets through the: Three.Geometry() library.
To do this, we use THREE.LineDashedMaterial that represent the material for the laser bullets, the bullets are dash lines, but this lines are visible for a short time.

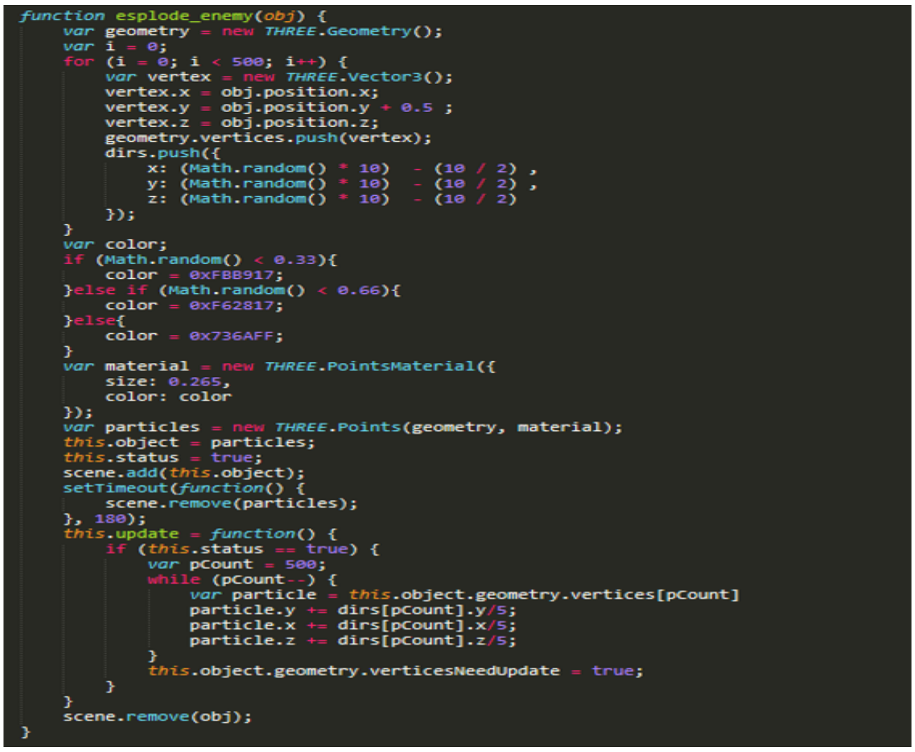


When the starship shoots a bullet is created this Geometry and is it called the ***hitShot()*** function where we used the Raycaster class which is designed to assist with raycasting. The raycasting is the use of ray–surface intersection tests to solve a variety of problems in 3D computer graphics and computational geometry, it is used for mouse picking (working out what objects in the 3d space the mouse is over) among other things. We implemented Raycaster in the *hitShot()* function that takes in input mouse position in normalized device coordinates (e.g ***mouse.x = (event.clientX/window.innerWidth) * 2 - 1; mouse.y = - (event.clientY / window.innerHeight) * 2 + 1***),then we set a direction for the created ray starting from the camera position and ending to the mouse position (e.g ***raycaster.setFromCamera(mouse,camera))***. Right now, each object that intersects the given ray will be returned as output by the ***intersectObjects(scene.children,true)*** function, of these objects we consider only the first object intersected and so we check if it is one of the enemy starships making a comparison the name of the object. In this case we remove the intersected enemy starship, level by level the enemies increase, so the controls on the intersected object also increase. Let's see a snippet of this function:



```
function hitShot( x,y ) {
    if (canShot){
        mouse.x = x;
        mouse.y = y;
        raycaster = new THREE.Raycaster();
        raycaster.setFromCamera( mouse, camera );
        var intersects = raycaster.intersectObjects( scene.children, true);
        if (level == 1){
            if( intersects.length > 0 ) {
                var firstObjIntersected = intersects[0].object;
                if ( enemystarship.enemystarship1.name === firstObjIntersected.parent.name ) {
                    enemystarship.enemystarship1.health -= starship.damage;
                    shooting = true;
                    if (enemystarship.enemystarship1.health <= 0){
                        parts.push(new esplode_enemy(enemystarship.enemystarship1.model));
                        enemystarship.enemystarship1.appeared = false;
                        enemystarship.enemystarship1.model.position.z = -1000;
                        enemystarship.enemystarship1.model = enemystarship.originalenemy1.model.clone();
                        scene.add(enemystarship.enemystarship1.model);
                        clock_money.start();
                        shooting = false;
                        level = 2;
                        document.getElementById("levelDiv").innerHTML = "LEVEL: " + level;
                        return;
                    }
                    loadDamagedShip(enemystarship.enemystarship1.model, enemystarship.enemystarship1.health);
                }
            }
        }
```

When our starship is hit we call the ***loadDamagedShip()*** function which creates the "damaged" effect. This is possible because we have created damaged models of some parts of our model, then after a hit we loaded the damaged parts.
Instead, when an enemy starship is hit and its health (they are 5 for basic enemies, 10 for the final enemy) ends up we recall the ***explode_enemy()*** function. Inside we use Three.Geometry, we create a Three.Vector3 object to give the starting point of the explosion and we give it a direction. Next we create a Three.Points object and through it we give a shape to the explosion. Below let's see the code of this function:



```
function esplode_enemy(obj) {
    var geometry = new THREE.Geometry();
    var i = 0;
    for (i = 0; i < 500; i++) {
        var vertex = new THREE.Vector3();
        vertex.x = obj.position.x;
        vertex.y = obj.position.y + 0.5 ;
        vertex.z = obj.position.z;
        geometry.vertices.push(vertex);
        dirs.push({
            x: (Math.random() * 10)   - (10 / 2) ,
            y: (Math.random() * 10)   - (10 / 2) ,
            z: (Math.random() * 10)   - (10 / 2)
        });
    }
    var color;
    if (Math.random() < 0.33){
        color = 0xFBB917;
    }else if (Math.random() < 0.66){
        color = 0xF62817;
    }else{
        color = 0x736AFF;
    }
    var material = new THREE.PointsMaterial({
        size: 0.265,
        color: color
    });
    var particles = new THREE.Points(geometry, material);
    this.object = particles;
    this.status = true;
    scene.add(this.object);
    setTimeout(function() {
        scene.remove(particles);
    }, 180);
    this.update = function() {
        if (this.status == true) {
            var pCount = 500;
            while (pCount--) {
                var particle = this.object.geometry.vertices[pCount]
                particle.y += dirs[pCount].y/5;
                particle.x += dirs[pCount].x/5;
                particle.z += dirs[pCount].z/5;
            }
            this.object.geometry.verticesNeedUpdate = true;
        }
    }
    scene.remove(obj);
}
```

Later we added controls to handle shots against the planets. Then we created a function similar to ***esplode_enemy()*** function for the explosion of the planets. We have made the necessary changes to this new function.
Below are some depictions of the model:

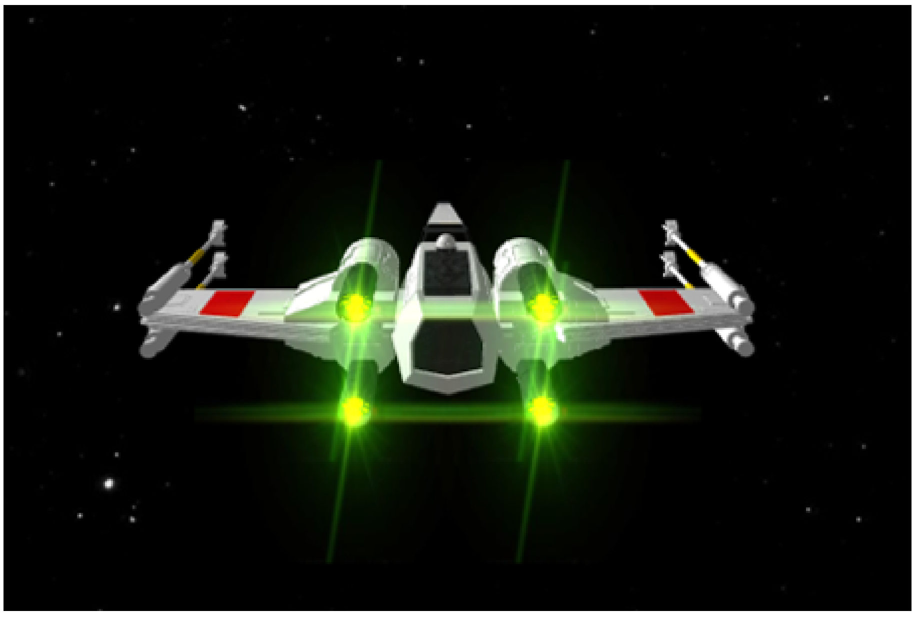
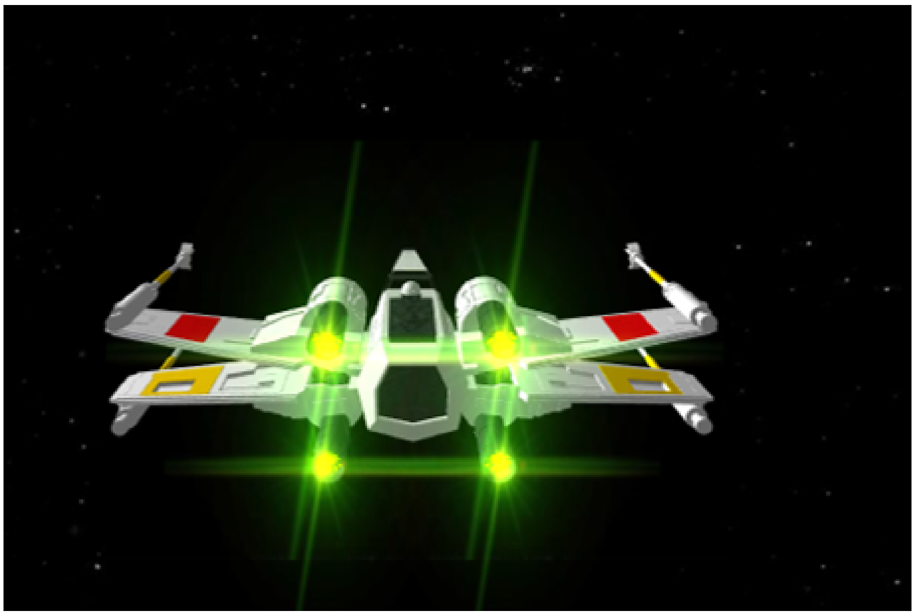



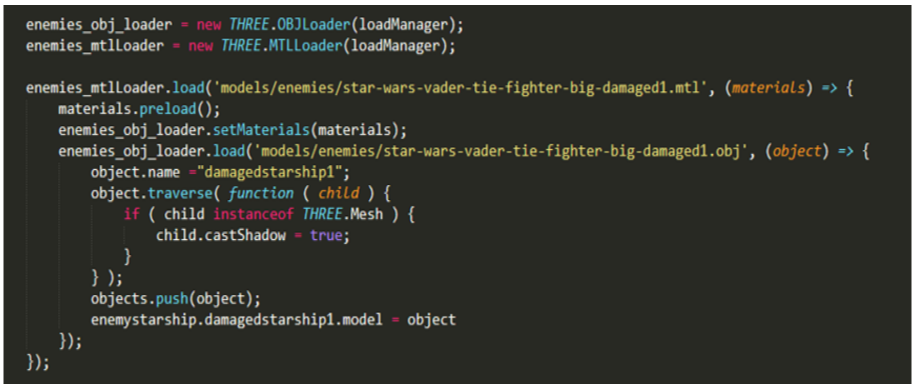
Figure 4: Starship Model

- ***ENEMIES STARSHIP***

  The Model of the enemies star-ships are composed by five equal star-ships and a final star-ship. Each starship has as parameters the model used to create the object, the name and the health.

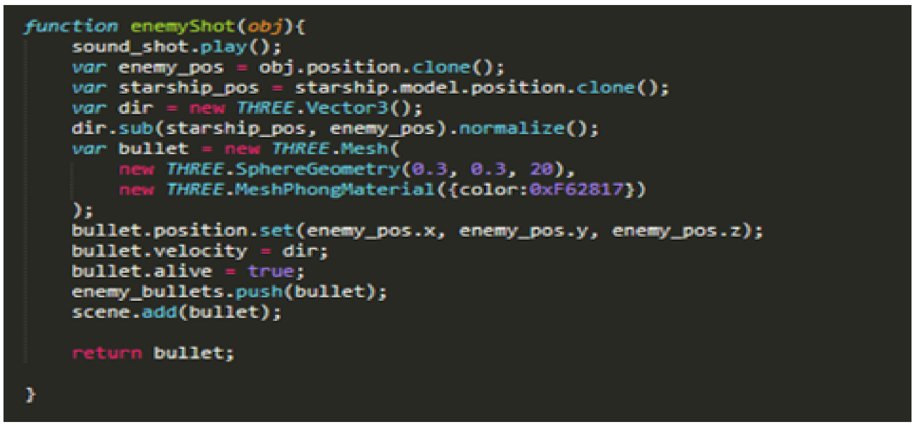
```
enemystarship = {
        originalenemy1 : {
                model:null,
                name:"enemystarship1",
                health: 5
        },
```

  The first thing to do is import the .OBJ file using the MTLLoader.



```
enemies_obj_loader = new THREE.OBJLoader(loadManager);
enemies_mtlLoader = new THREE.MTLLoader(loadManager);

enemies_mtlLoader.load('models/enemies/star-wars-vader-tie-fighter-big-damaged1.mtl', (materials) => {
    materials.preload();
    enemies_obj_loader.setMaterials(materials);
    enemies_obj_loader.load('models/enemies/star-wars-vader-tie-fighter-big-damaged1.obj', (object) => {
        object.name ="damagedstarship1";
        object.traverse( function ( child ) {
            if ( child instanceof THREE.Mesh ) {
                child.castShadow = true;
            }
        } );
        objects.push(object);
        enemystarship.damagedstarship1.model = object
    });
});
```

  Each enemies starship has an animation handled with the ***animateEnemy()*** function. When we start the game, the position of the enemies has coordinate with z equal to a very small negative number (less then the far value of the camera) so that the star-ship is not visible by the player. Through the function: function loadEnemies() in each step of the level the enemy star-ship increases the z position and in this way for the player is like the star-ship is approaching to the enemy star-ship.
  When the loadEnemies() finishes it will start the animateEnemy() function and so the battle too. Each enemy has a different animation and in addition to moving in space can shoot a bullet in direction of the star-ship (of the player). The first function to do this is : function enemyShot(obj). In this function in addition to the sound for the shot there is :



```
function enemyShot(obj){
    sound_shot.play();
    var enemy_pos = obj.position.clone();
    var starship_pos = starship.model.position.clone();
    var dir = new THREE.Vector3();
    dir.sub(starship_pos, enemy_pos).normalize();
    var bullet = new THREE.Mesh(
        new THREE.SphereGeometry(0.3, 0.3, 20),
        new THREE.MeshPhongMaterial({color:0xF62817})
    );
    bullet.position.set(enemy_pos.x, enemy_pos.y, enemy_pos.z);
    bullet.velocity = dir;
    bullet.alive = true;
    enemy_bullets.push(bullet);
    scene.add(bullet);

    return bullet;

}
```

This part is used to determinate the position of the enemy star-ship,the direction in which the bullet will be fired and for the creation of the bullet. And in the end it is set the position from where the shot starts and the direction that the bullet will take.
This is only the function for the creation of the bullets but this function is called through the shotResponse() function, when an enemy is appeared, it starts to shot you.
To determinate if the enemy hits you, as the principal starship, we implemented the THREE.Raycaster() library. In this way if the bullet intersects the starship the starship.health value decrease.
When the starship.health value is less or equal to zero the game finish and appear a write: GAME OVER, so the player can restart the game by pressing a button on the keyboard.

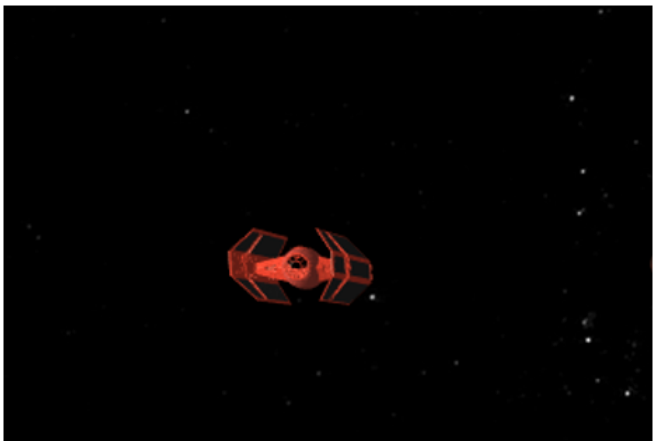


Figure 5: Basic Enemy

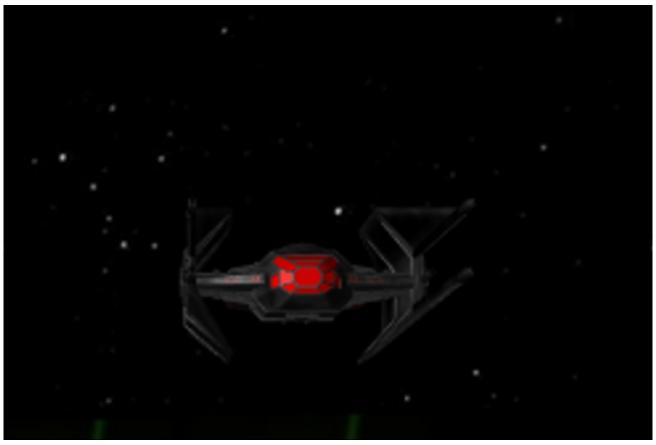
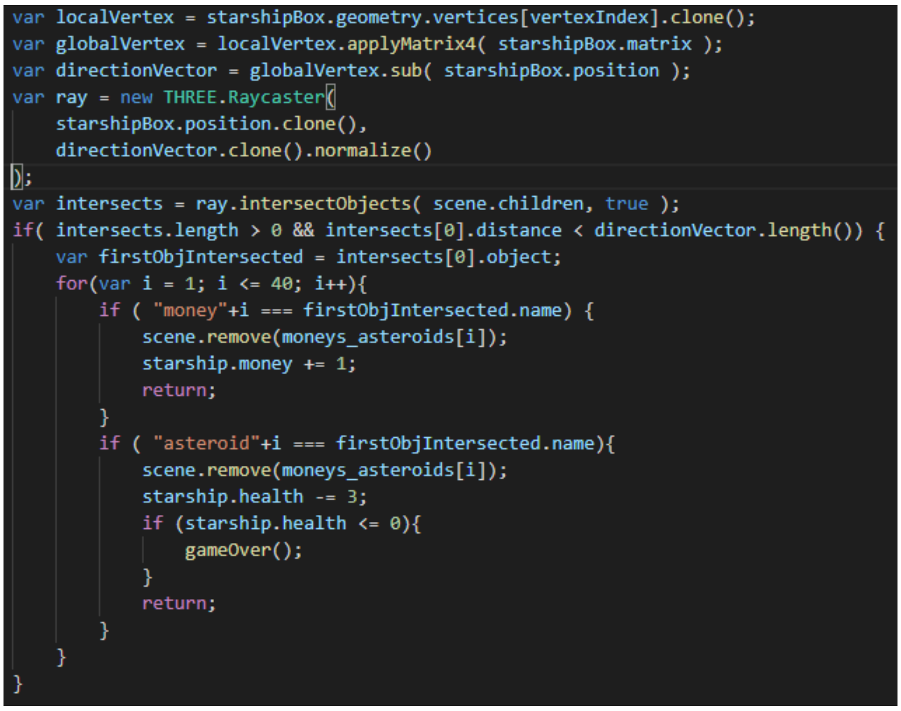


Figure 6: Final Enemy

- ***MONEYS AND ASTEROIDS*** After the first and the second level, our starship has some moments of break from the enemies and the player can restore its health or improve damage by collecting money. So there are two sequences of moneys introduced between two levels that can be collected by the starship by flying into them.
  So let's now talk about technical aspects: there are not only moneys, but interleaved with them there are some asteroids very dangerous for the starship's health and so a player should take care to avoid them while collecting moneys. We have two types of objects:

  - **Moneys**: they are build by defining a *CylinderGeometry* with a ray of 2 for both the faces and high 1, the minimum. Finally we put 64 as the number of triangular faces around the circumference of the cylinder to achieve a cylinder as smoothed as possible. About the material we apply to each money a procedural texture which applies a gradual shade to the money and mesh it with a pure color. We defined the material as a PhongMaterial so that it can reflect lights.
  - **Asteroids**: similiar things can be said for the asteroids: they are spheres with a ray of 2 and their material is mapped again with a procedural texture, this time a ranomized one to reach an effect of an irregular surface.

  Of course moneys and asteroids are handled as an array of objects and interleaved each other. In total they are about 40, and they are positioned along an euclid spiral to create some difficulty to players when trying to catching them.
  Let's now focus on the interaction between these objects and starship and how the collision is realized.
  Again, we use Raycaster library to get collisions through intersections. We define a function that is put in the animate and so it is continuosly called and to improve raycasting we define a geometryBox around the starship in order to better simulate money drop of the starship. From this helper box we take some vertices of its faces (unlickily we can't take every one because it will make too slowly the rendering) and from each of them we create a ray starting from the starship position and with a direction set by the vector between the starship position itself and the vertex under consideration.
  After setting up the ray we take the output of the intersectsObjects library function which return a list of all objects intersected in the chosen direction from the source point. Of these object we only take the ones which distance is less than the chosen direction vector length.
  Finally, by taking the first intersected object, we check if it is a money, and in this case we update moneys earned, of it is an asteroid, and in this case we decrease starship health.
  In the following, a code snippet of the above described function and then an image of the described objects.



```
var localVertex = starshipBox.geometry.vertices[vertexIndex].clone();
var globalVertex = localVertex.applyMatrix4( starshipBox.matrix );
var directionVector = globalVertex.sub( starshipBox.position );
var ray = new THREE.Raycaster(
    starshipBox.position.clone(),
    directionVector.clone().normalize()
);
var intersects = ray.intersectObjects( scene.children, true );
if( intersects.length > 0 && intersects[0].distance < directionVector.length()) {
    var firstObjIntersected = intersects[0].object;
    for(var i = 1; i <= 40; i++){
        if ( "money"+i === firstObjIntersected.name) {
            scene.remove(moneys_asteroids[i]);
            starship.money += 1;
            return;
        }
        if ( "asteroid"+i === firstObjIntersected.name){
            scene.remove(moneys_asteroids[i]);
            starship.health -= 3;
            if (starship.health <= 0){
                gameOver();
            }
            return;
        }
    }
}
```

Figure 7: Money and Asteroid collision

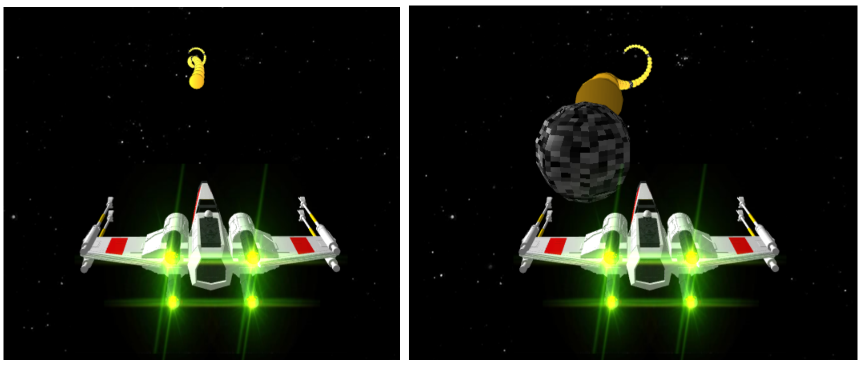


Figure 8: Money - Asteroid

- ***PLANETS*** In this section we will talk about the planets: their composition, their movement and their interaction with other objects.

  Let's start with the physical composition of the object.
  After declaring the object, we have created the planets data. Each planet has important information that characterizes it, like rotation rate, distance from axis, name, texture, size, segments.
  Once we have defined all the useful parameters for the objects, we have created the planet using the loadTexturedPlanet function that takes in input the data of the planet(previously listed) the starting position (x, y, z) and the material type. In detail, in this function are defined: the spherical shape, the final material, the size and texture of each planet.
  To randomly manage the size and textures, a function has been implemented that deals with executing the vector shuffle (function shuffle(myArray,myLen)). Two vectors have been created, one with all possible dimensions and one with all types of textures. At each starting of the game, a size and a texture is randomly selected for each planet.
  The texture of the planets are all images of various formats (jpg, png). They are all placed in the images folder and, to facilitate management, they have been numbered progressively from one to twenty-six. These textures are applied to the sphere that represents the planet in the loadTexturedPlanet function.

  Now let's start with the movement of the object.
  Each planet has two types of movement: a translation in space and a rotation on itself (movement on the axes).
  As for the translation on space, to give an idea of fluidity to the game we have set the same speed to all the planets.
  As far as the rotation on itself is concerned, all the planets rotate with respect to the x axis with the same intensity, while with respect to the y axis they have similar but slightly different coefficients. This choice was made to give the game a sense of uniform and constant movement.

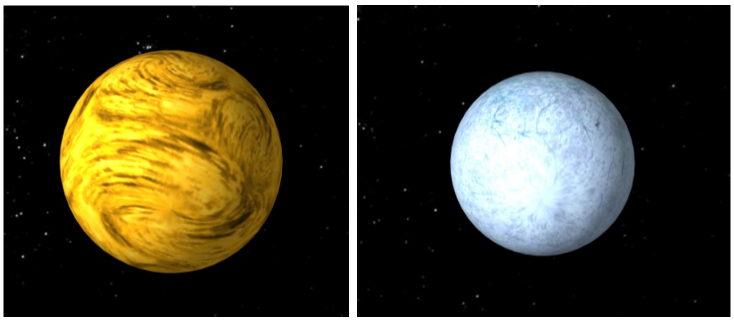
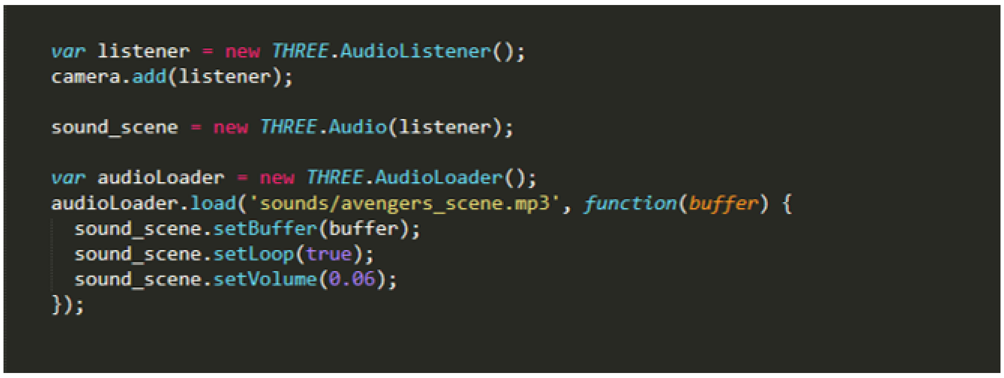


Figure 9: Examples of planets

5 User's interactions

5.1 Audio

To make the user experience as realistic as possible, we decided to include sound in the interaction of starship with the scene. We gave a background sound, a lighting sound of the starship, a sound to the shots of starships and enemies. This has been possible thanks to Three.AudioLoader, we have created an AudioListener and added it to the camera. Then we have created a global audio source Three.Audio and finally we have loaded a sound and setted it as the Audio object's buffer.

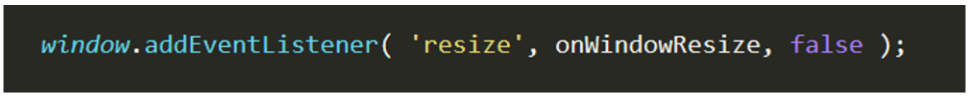


```
var listener = new THREE.AudioListener();
camera.add(listener);

sound_scene = new THREE.Audio(listener);

var audioLoader = new THREE.AudioLoader();
audioLoader.load('sounds/avengers_scene.mp3', function(buffer) {
  sound_scene.setBuffer(buffer);
  sound_scene.setLoop(true);
  sound_scene.setVolume(0.06);
});
```

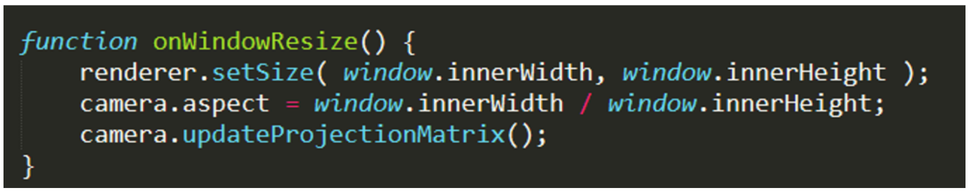
We have inserted this snippet code into the ***loadModels()*** function.
For the scene sound we set ***sound_scene.setLoop(true)*** to make the playback loop, obviously it starts at the beginning of the game so we called ***sound_scene.play()*** in the ***initGame()*** function. The same logic applies to the other sounds created, we inserted their loading in the ***loadModels()*** function and added their playing in the appropriate functions.

5.2 Resizing the window

To make the user experience more enjoyable we added event handlers for resizing the window. On the window listener resize we refer to a function created by us.



```
window.addEventListener( 'resize', onWindowResize, false );
```

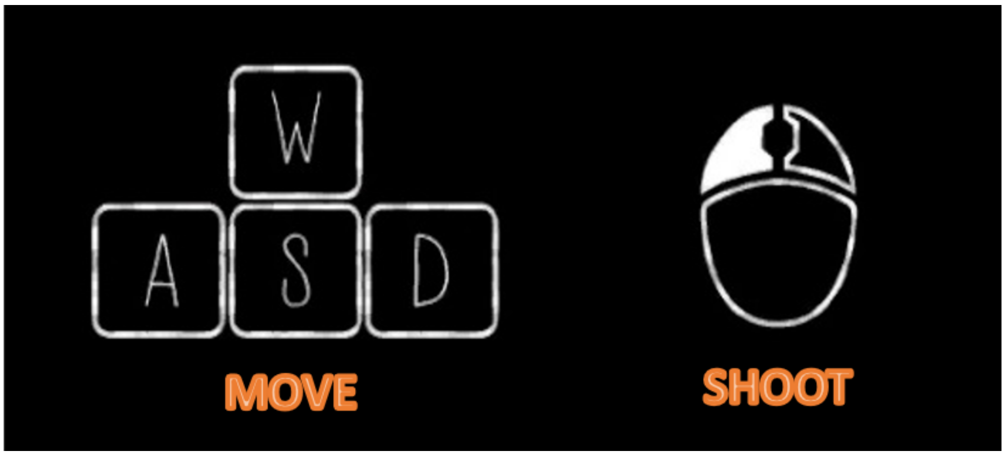


```
function onWindowResize() {
    renderer.setSize( window.innerWidth, window.innerHeight );
    camera.aspect = window.innerWidth / window.innerHeight;
    camera.updateProjectionMatrix();
}
```

Since the canvas is fullscreen, here we simply use the window's client area. The camera also need to be notified of the changed aspect ratio. This is important or everything looks squashed. Since the camera can not detect that one of it's properties changed, we need to call ***updateProjectionMatrix()*** function.

5.3 Commands

To manage the starship and to enjoy the experience, the user can interact with the keyboard with keys chosen by us.



We also added the ”P” key to quit the game and the ”R” key to restart the game.

6 Bibliography

References

[1] **Three.js documentation**
https://threejs.org/docs/.

[2] **Three.js examples**
http://stemkoski.github.io/Three.js/.

[3] **Planets textures**
http://robinsonprogramming.com/solar_system.php.

[4] **Background scene**
https://codepen.io/.

[5] **CSS animation**
https://skillsandmore.org/animazioni-css-keyframe/.

[6] **Starships models**
https://free3d.com/3d-model/x-wing-88229.html.
https://clara.io/view/20d6eb15-0a46-4094-8eaf-fb4d9ad458c9.
https://sketchfab.com/3d-models/tievn-silencer-4d6d3793ce7f432c9141b14cb7d7c512.