Interactive Graphics

Videogame Project: **"Sea Fighters"**

Damiano Brunori, Lorenzo Saiella and Alessandro Trapasso

Sapienza Università di Roma

Master in Artificial Intelligence and Robotics/Computer Science

1 Project Overview

We have created an arcade videogame settled in the sea. For this task it has been used mainly the *three.js* library. In the *three.js* library there is not a physical engine which allows us to manage the gravity and the various collisions, and thus, in order to achieve this target, we have used the *Raycaster* class available in the *three.js* library. For the environment we have imported some *.obj* files regarding the islands, the player ship, the enemy ship and the bullets. In addiction we have created others enemies which are based on hierarchical models. The main idea behind the game mode is a very simple one, i.e. you have to destroy all the enemies ships by avoiding your *health bar* get reset.
There are some main different functions that we want to highlight and which are related to the various parts of the game:

init();
renderer();
update().

Now we will show in detail the aforementioned functions.

2 Function *init()*

2.1 World Game

We have created a scene and add a perspective camera to this scene; the camera is initially settled with an angle of 45 degrees behind the ship of the player. The view angle of the camera has been set similar as possible to the human visual field, thus we have used these main following parameters values:

- **view angle**: 45 degrees;
- **near**: 0.1;
- **far**: 20000.

In order to control the camera orbit around our target (i.e. the ship of the player) we have used the class *OrbitControls*; we have also set a distance range from the player's ship which is between 80 and 400 pixels (this value can be changed by using the mouse wheel). For the environment's lighting has been used a white directional light starting from the position of the Sun which is created by the function *updateSun()*.
The water element has been realized by using the class *water*; the water is 'placed' on a *PlaneBufferGeometry*, which is a class representing a 3D plane of 20000x20000 pixels. For this task we have used a texture, a scalar distortion for the reflection and a rotation determining the movement of the waves.
We have also created the sky by setting the tilt distance and the Azimut; even the audio has been included with the function *audio()*.
To load the *.obj* files we have created the function *Object3D()* which load both the *.obj* and the *.mtl* files; the latter is made up by one or more material and each of these include the color, the

texture and the reflection map of individual materials.

2.2 Game's Element

In this section we will describe the various objects included in the scene.

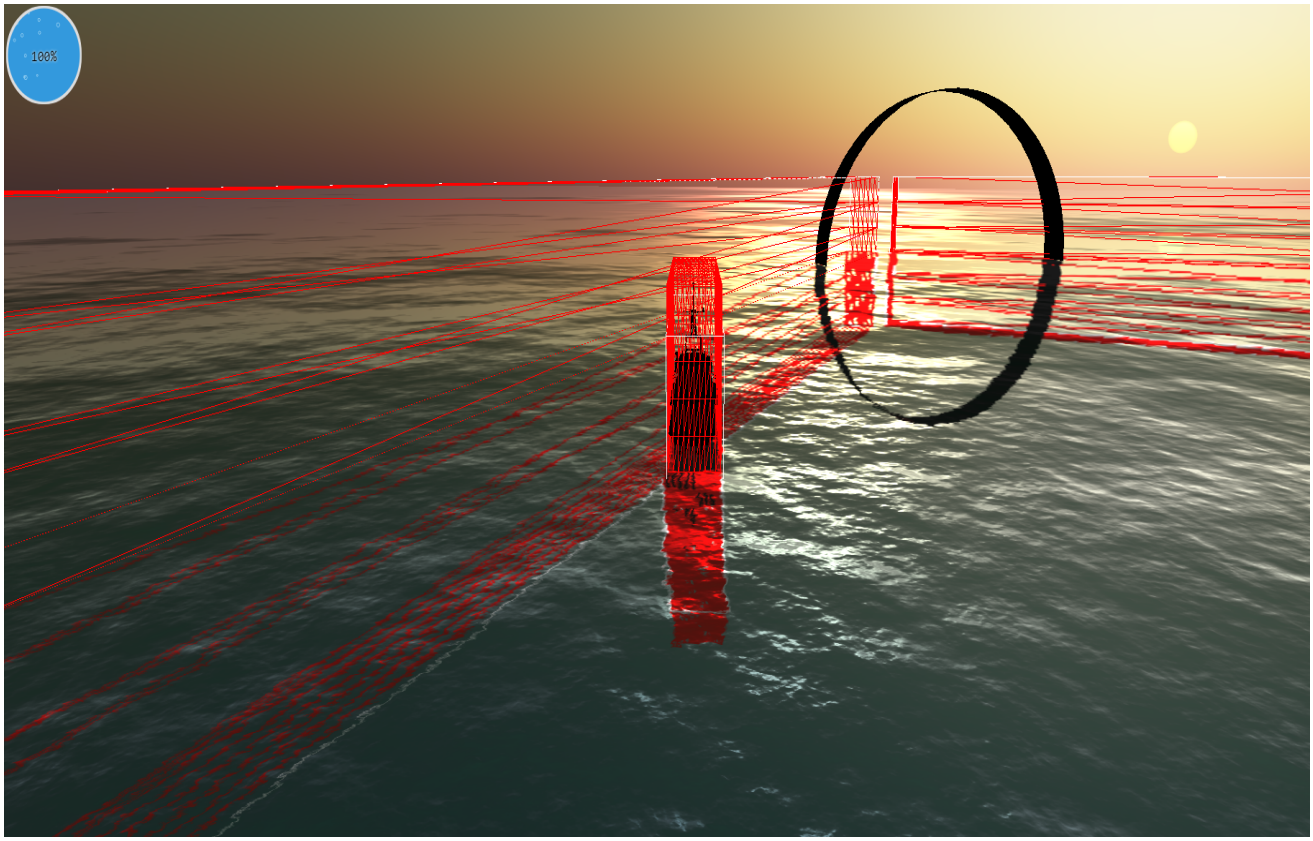

Fig.1 The StarGate standing in front of the player's ship.

2.2.1 Fixed Objects

Let's start by analyzing the fixed objects like the *stargates*, the *islands* and the *wall boxes*. The *islands* have been loaded with the function *Object3D()* and then we have rotated, translated and scaled them in order to fit all these objects to our world's game.
Regarding the *stargate* (Figure 1) we have created a function *MagicPort()* which generates four ports, each of which is made up by an empty cylinder which allows us to move the player's ship from a stargate to another one placed in the opposite side. These cylinders have been realized with the class *ThreeBSP* and they are composed by a geometry and a material which are merged together. The 'magic ports' are obviously placed at the four angles of the game's borders in such a way they are specular, facing each other.
The wall boxes delimits the game's space of action, the islands, the enemies and the bullets. These boxes have been created by using the class *Mesh* which contains the geometry and the mesh basic material. We have set the material in such a way that the containers (i.e., the boxes used for the collision detection) result not visible.

2.2.2 Animated Objects

Here we will describe the animated objects like *arms* and *enemies*. Each arm is composed by four cylinders and three spheres (the latter have been used like elbows and add it between the cylinders). Each arm is also made up by four cylinders and three balls; the balls have been used as elbows inserted between the joints of the various cylinders. The four arms (cylinders) have been realized by using the function *THREE.CylinderGeometry()*, thanks to which the circumference, the length, the height and everything related to the cylinder geometry have been set. The material of each cylinder has been created by setting *var cyl_ material = new THREE.MeshBasicMaterial()*. We have then applied a matrix to the geometry of the cylinder, so as to modify the rotation point by changing it from the default value (i.e., the center of the cylinder) to the desired one (e.g., *(lunghezzaBraccio/2*). Thanks to the class *cylGeometry.applyMatrix*, it was possible to rotate the cylinder around its extremity; we have used also a mesh with the geometry and the material of the cylinder. After the first phase of creating the object, it was necessary:

- translate the arm;
- rotate the arm (these first two steps have been done by performing calculations on the *x, y, z* axis);
- insert each cylinder in the desired position.

Let's take as an example what has been done for the left arm (*arm -> cylinderSx3*); in order to insert it correctly in the world, it was necessary to perform a translation on the *y* axis of the cylinder (this translation is half the length of the length of the of the second cylinder. Then we have added half the length of the third cylinder, and finally we have performed a rotation on the *x* axis of *angleBraccio3*, namely a rotation of *-Math.PI/2*. Each cylinder has been added to another cylinder through the *.add* method.
The other arms (cylinders) have been created in a similar way, by recalibrating each time the rotation and translation angles on the various axis. Finally we have added three spheres which fit into the three joints of the arms in oder to make it more harmonious, (we have realized all of this by setting *(Variable) = new THREE.Mesh(new THREE.SphereGeometry)*

3 Function *update()*

In this function we have created a variable *delta* to keep track of time and a variable *time* with *Date.now()* method. These variables have been created to set the speed of the ship navigation (200 pixels at second) and rotation (*PI/5* per second).

We have set the command keys to play through four *if-clause* in the follow way:

- by pressing the *W* key on the keyboard, the ship makes a positive translation on the z axis;
- by pressing the *A* key on the keyboard, the ship makes a rotation on its y axis of a positive angle;
- by pressing the *S* key on the keyboard, the ship makes a negative translation on the z axis;
by pressing the *D* key on the keyboard, the ship makes a rotation on its y axis of a negative angle;
- by pressing the *SPACEBAR* key on the keyboard, the ship shoot in the desired direction.

With this latter function the bullet will be created (the variable *canShoot* indicates the amount of time which is between the shoots). A cube is added to each bullet in order to detect collisions. The bullets and the cube are push in two different arrays, i.e. *bullets[]* and *collidableMeshList1[]*. Thus we equate the position and the respective rotations of the missile to that of the boat. We have also used two *if-clause* which allows us to rotate and guide the bullets in the correct direction.
Inside the *turret()* function we create two cylinders and a sphere to which apply the class *THREE.Matrix4()*. In this way we have been able to move the pivot point of each cylinder so you can translate correctly a cylinder accordingly to the position of the other cylinder. The sphere is settled between the two cylinders like a joint.

To compute the ray of action of the turret, we have calculated the distance between the position of our ship and that of the turret itself. When the boat enters inside the ray of action of the turret, the function *nemiciTorretta()* is recalled. For this function, the same logic used in the function*myBullet()* was applied.

The functions *nemiciTorretta()*, *enemyShoot()* and *nemici()* are very similar to each other and they differ only for the position, the direction and the rotation of the bullets.

3.1 Collisions Detection

We have used the function *WallsLimits()* to detect collisions between our boat and the walls; we have also set a collision check (i.e. a collision is verified when our boat intersects the wall which delimits the game space). When the boat collides with the wall, it moves itself back of 40 pixels on own axis.

There is also another function, that is *teletrasporto()*, which allows the boat to teleport to the 'stargate' opposite to that one in which you entered. When the collision is verified, the position of the boat is changed. In all the figures shown in this paper you can see a set of red lines which allows us to manage the different collisions inside the videogame.

3.1.1 Raycast Collision Detection

For the collision between our boat and the islands, we have increased the projection of the rays of our cube in order to have more accuracy in the collision detection. Our raycasting code casts a ray from the *MovingCube*'s position towards each of its vertices. Whenever the ray intersects something, it is returned the intersected object along with the distance from the *MovingCube*'s position at which the object was found. If the distance to the collision is less than the distance to the vertex, a collision happened inside the cube.
By *collisionResults[1].face.normal* we obtain the the normal of the face in which the collision occurs; this normal vector will be 1 or -1 respect to the positive and negative axis of z and x. Every time the ship impacts a cube's face of a game's element, it moves by 30 pixels back in the normal direction of the stricken element's face.
You can see better how this process works looking at the Figure 2.

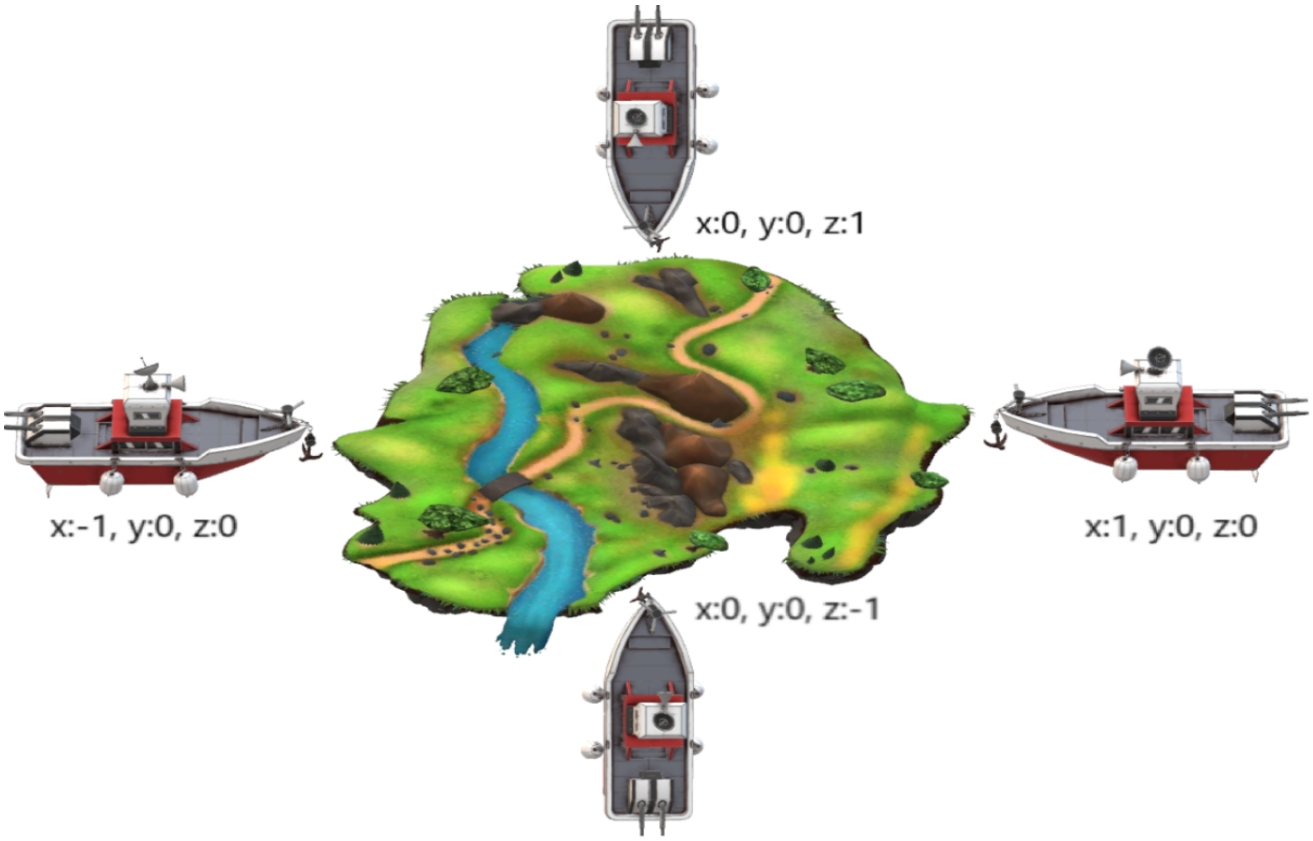

Fig.2 It is shown here how it is saved the normal vector when the ship strikes an island.

We have done the same operation for the collision with the bullets; when the collision is verified, then a particles explosion occurs, and all the bullets fired by the relative enemy are removed from the array containing them and from the scene. The explosion is a set of a random moving cubic particles which are fired from a vertex of the stricken object. This particles explosion survives for 2 seconds.

3.1.2 Enemies AI

The enemies AI regards the ability of the enemies to identify our ship. We have used the same raycast's function applied previously to detect collisions, but in this case the collisions occurs when our boat enters in the enemy ray of action; when all this happens, then the enemies detect our position and shoot the bullets towards us as long as we are within their range of action. We have also used the function *EnemyShoot()* to fire in the correct direction.
About the movement of the enemy ship, we have used the *path* function which creates a curve line made up by 200 different points selected from rectangular pseudo-random areas; these areas have been chosen in such a way that they not fill the surface covered by the islands. The enemy ship will move by following the route traced in this manner, slowing down and starting to fire when the player's ship is detected. You can have a better overview of all this by looking at the Figure 3.

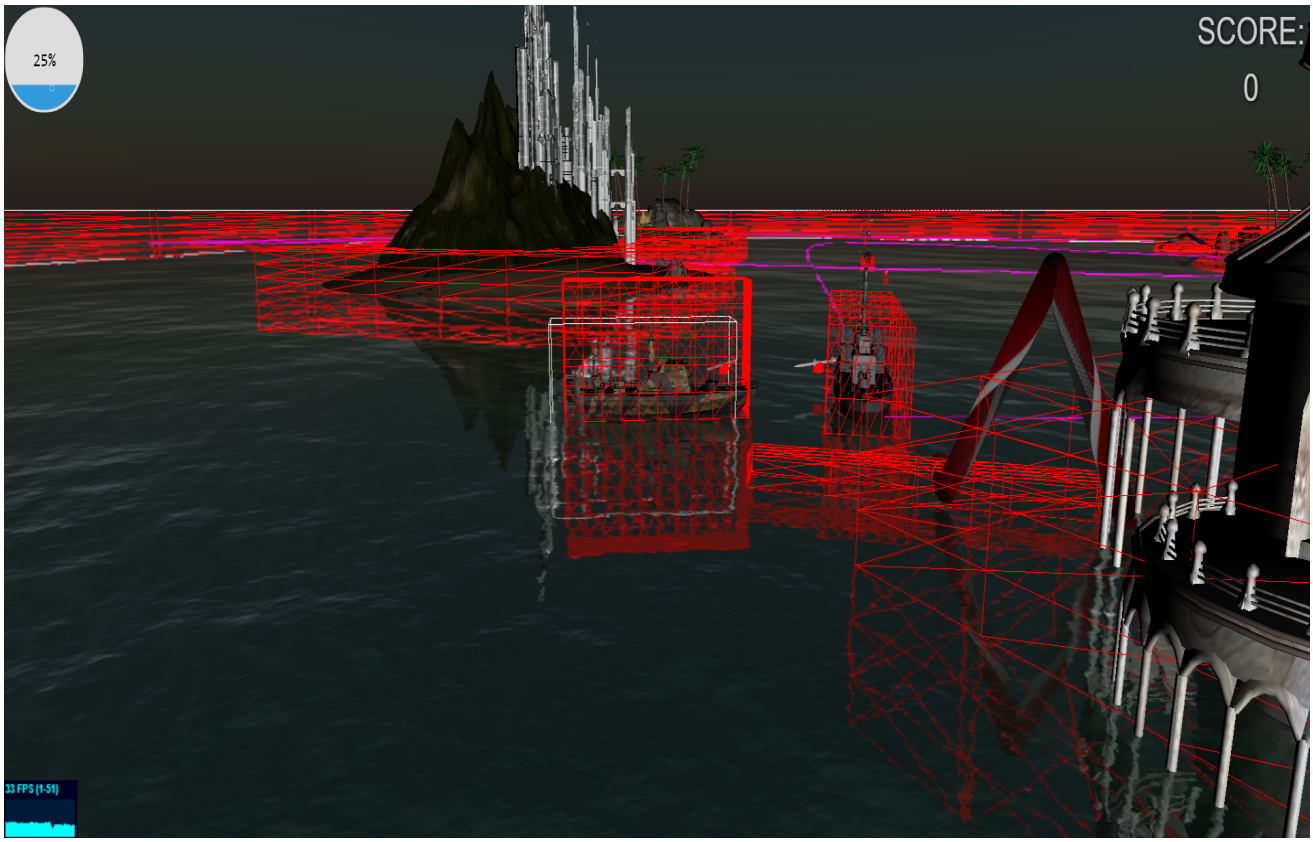

Fig.3 It is possible to see the enemy ship (on the right) which shoots the player's ship (on the left) standing in the enemy range of action; the route followed by the enemy ship is marked in violet.

4 Function *renderer()*

The pointing direction of the turret is continuously computed and updated according to the position of the our ship; in this way we were able to create a vector in which we save the new direction towards which the turret's cylinder has expected to be oriented. Thus the turret is able to point towards our ship.

We want to mention now the relevance of the variable *pCOUNT* which contains the number of particles which have been created as a consequence of the explosion. This number of particles is updated through a *while-cicle*.

As far as the animation of the cylinders is concerned, we have set a variable *count* which allows us to change the sign to the speed rotation of the *cylinder2* and *cylinder3*; this makes the arm stretch and fold on itself. To make this task more natural, the rotation speed of the *cylinder3* is twice that of the *cylinder2*.

Finally, we want to say something about *renderer.render (scene, camera)*. We have created a loop in such a way that the renderer draws the scene every time the screen is updated (60 times per second FPS). This gives the user the advantage of not wasting his processing power when he

navigates to another browser tab.

5 Gameplay

The user is on a boat in the middle of the ocean, surrounded by islands and enemies. The aim of the game is to kill the enemies before being killed. You earn points every time you hit an enemy. The user can make the ship move by pressing the keyboard keys (W, A, S, D) and shoot the missiles by pressing 'space' (see *Chapter 3* for more detail about the commands). The enemies are divided into three categories, namely enemy boats, turrets and mechanical arms. Come into conflict when you enter an enemy's range of action.

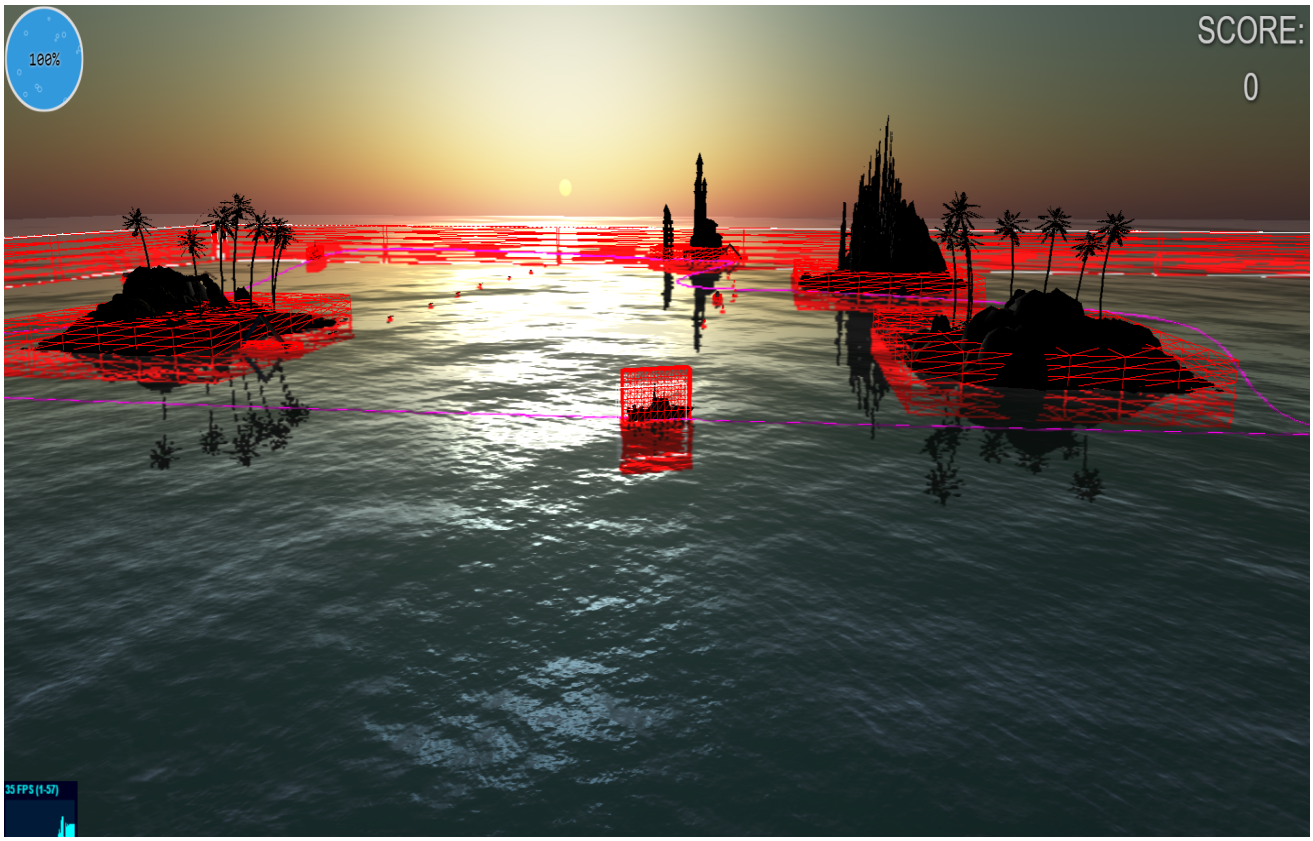

Fig.4 An overview of all the elements appearing in the game (islands, missiles, ships, stargates, turret and arm).

6 List Of The Libraries And Tools Used But Not Developed By The Team

Libraries:

- three.js;
- three.min.js;
- OrbitControl.js;
- Water.js;
- Sky.js;
- Detector.js;
- stats.min.js;
- THREEx.KeyboardState.js;
- OBJLoader.js;
- DDSLoader.js;
- MTLLoader.js;
- ThreeCSG.js;
- ParticleEngine.js;
- loading-bar.js;
- loading-bar.min.js.

Tools:

- Blender;
- Bootstrap.