Interactive Graphics

Hit the target

Ludovico Ottobre 1712005

Gianmarco Evangelista 171181

Contents

1. Introduction

The project chosen to satisfy all requirements is a video game in which you can simulate the possibility to shoot a target in several modes. The user experience consists in to accessing in three modality in order to hit a target.

Every choice gives to the user the possibility of being in a different background scene maintaining the main idea of the game. In particular we can find three different situation and each one will have a different execution, animation, background and models.

The user can control the gun in the first and second approach while in the third he can use the ball in order to drop targets.

In the first scene, the user's visual is in first person and he can manage a gun in order to hit a target as the traditional situation. Initially we have a number of hit equal to ten and at the end of the level, we will have the results: accuracy, perfect target hit, target hit or missed target. In order to achieve the second level, where we will have a different scene with a target between obstacles (trees and ducks), we have to achieve a minimum level of accuracy. If the second level accuracy is enough to pass the level difficulty, then the user can access to the third level in order to get the victory. In the third level we have more *trees and ducks* in order to create difficulties to user.

In the second scene, we don't have any type of level but we can see a different background scene, different gun and a different target animation. In order to see the stats, we have a timer and at the end of it**,** after 20 seconds, the system returns the number of hit, the number of target hit and **the accuracy.**
In particular in this scene we can find some planets as a background that are animated moving in place and a gold star that is the target to hit.

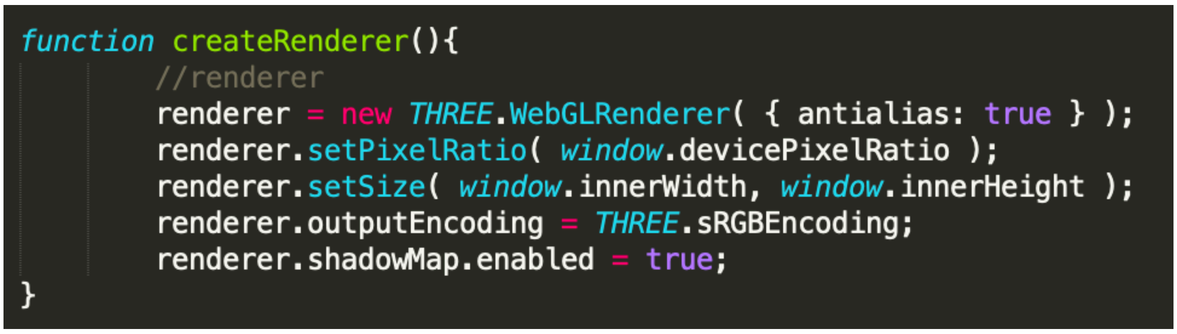
In the third scene we wanted to simulate the game of bowling through the effects of physics provided by physijs. During the game, the user can choose one of six levels, each of which provides a different arrangement of some obstacles that are placed between the ball and the pins, and can choose one of three balls. Moreover, the user can choose whether and when to retry the shot or whether to return directly to the home

In order to play, for each modality you have to use easily the mouse in order to hit the target and know that during the game by pressing the H key, you can return to home. All the others user’s interactions, are available in the menu on the right where you can set the light properties and choose whether to turn on or not the music.

2. Three.js: environment and scene

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 using WebGL.
In order to display anything with three.js we need three things: renderer, scene and camera. So we have created two functions:

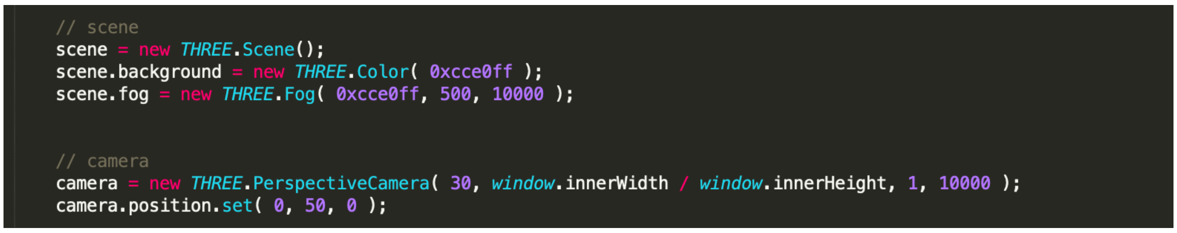
createRenderer( )



```
function createRenderer(){
        //renderer
        renderer = new THREE.WebGLRenderer( { antialias: true } );
        renderer.setPixelRatio( window.devicePixelRatio );
        renderer.setSize( window.innerWidth, window.innerHeight );
        renderer.outputEncoding = THREE.sRGBEncoding;
        renderer.shadowMap.enabled = true;
}
```

In addition to creating the renderer instance, we also need to set the size at which we want it to render our app. It's a good idea to use the width and height of the area we want to fill with our app - in this case, the width and height of the browser window. For performance intensive apps, you can also give setSize smaller values. For any heavy three.js app *setPixelRatio( )* is the right way in order to don't have a slow framerate. 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.

createScene( ):



```
    // scene
    scene = new THREE.Scene();
    scene.background = new THREE.Color( 0xcce0ff );
    scene.fog = new THREE.Fog( 0xcce0ff, 500, 10000 );


    // camera
    camera = new THREE.PerspectiveCamera( 30, window.innerWidth / window.innerHeight, 1, 10000 );
    camera.position.set( 0, 50, 0 );
```

Camera:
The first attribute is the field of view. FOV is the extent of the scene that is seen on the display at any given moment. The value is in degrees.
The second one is the aspect ratio. You almost always want to use the width of the element divided by the height, or you'll get the same result as when you play old movies on a widescreen TV - the image looks squished.
The next two attributes are the near and far clipping plane. What that means, is that objects further away from the camera than the value of far or closer than near won't be rendered. You don't have to worry about this now, but you may want to use other values in your apps to get better performance.
Next up is the renderer. This is where the magic happens. In addition to the WebGLRenderer we use here, three.js comes with a few others, often used as fallbacks for users with older browsers or for those who don't have WebGL support for some reason.

Scene:
Scene allow us to set up what and where is to be rendered by three.js. This is where we have placed objects, lights and cameras.

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.
In order to put a texture for each object, we have to use *TextureLoader*, a class for loading texture. This uses the ImageLoader internally for loading files.

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:
1. Ambient Light
2. Directional Light

The second one can be set through the control menu that appears during the game on the right. The user can decide to turn on or turn off the directional light, can choose whether to turn it on or not and can pause the general music. Through the function *initGui( ),* we have set the *param* and added them to *var gui* in order to manage each slider and button.

3. Shooting range

In order to implement this game, we have to use some functions and several models. It is divided in three different level, each one with a different difficulty and in order to achieve a good result is necessary that the user gets a specific accuracy values in general:
First Level > 0.5, Second Level >0.7, Third Level > 0.9.
We can divide the execution of the game deciding how many hits the user will have for each level and at the end of each one, he can decide whether to go on or stop himself.
The implementation of the gun has been done with the function *createGun( )* where we have used *MTLLoader( ),* a loader for loading an *.mtl* resource, used internally by OBJLoader that is a loader for loading a .obj resource and the OBJ file format is a simple data-format that represents 3D geometry in a human readable format as the position of each vertex, the UV position of each texture coordinate vertex, vertex normals, and the faces that make each polygon defined as a list of vertices, and texture vertices.
In order to load a texture, we have used *THREE.TextureLoader( ),* specifying what is the right path to load what we want. In order to manage in the right way the texture setting, it is important to know that we have used a hierarchical model and so for each child, we had to set the texture.
Thanks to *animateCloseHammer( )* and *animateOpenHammer( )*, it has been set the animation of the trigger during the function *fire( )* in order to give to user the feeling of reality.
In the same way we have created the ground, using *createGround( )* function we have loaded the right texture and repeat it in order to cover all the scene.

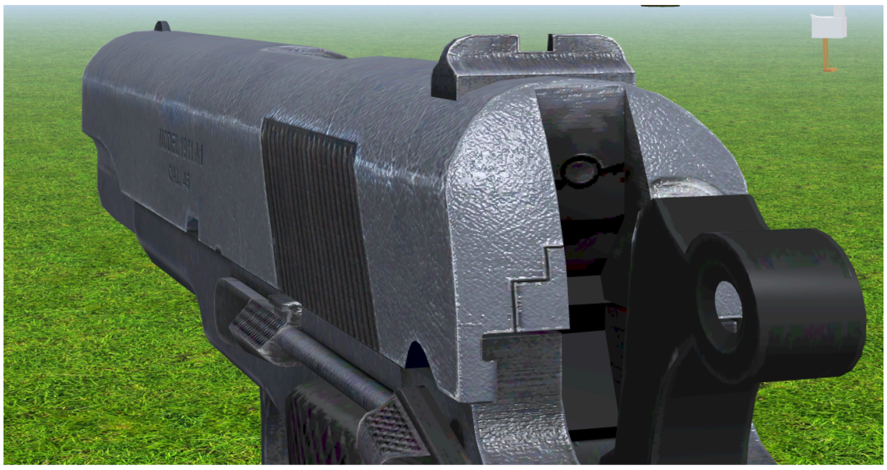
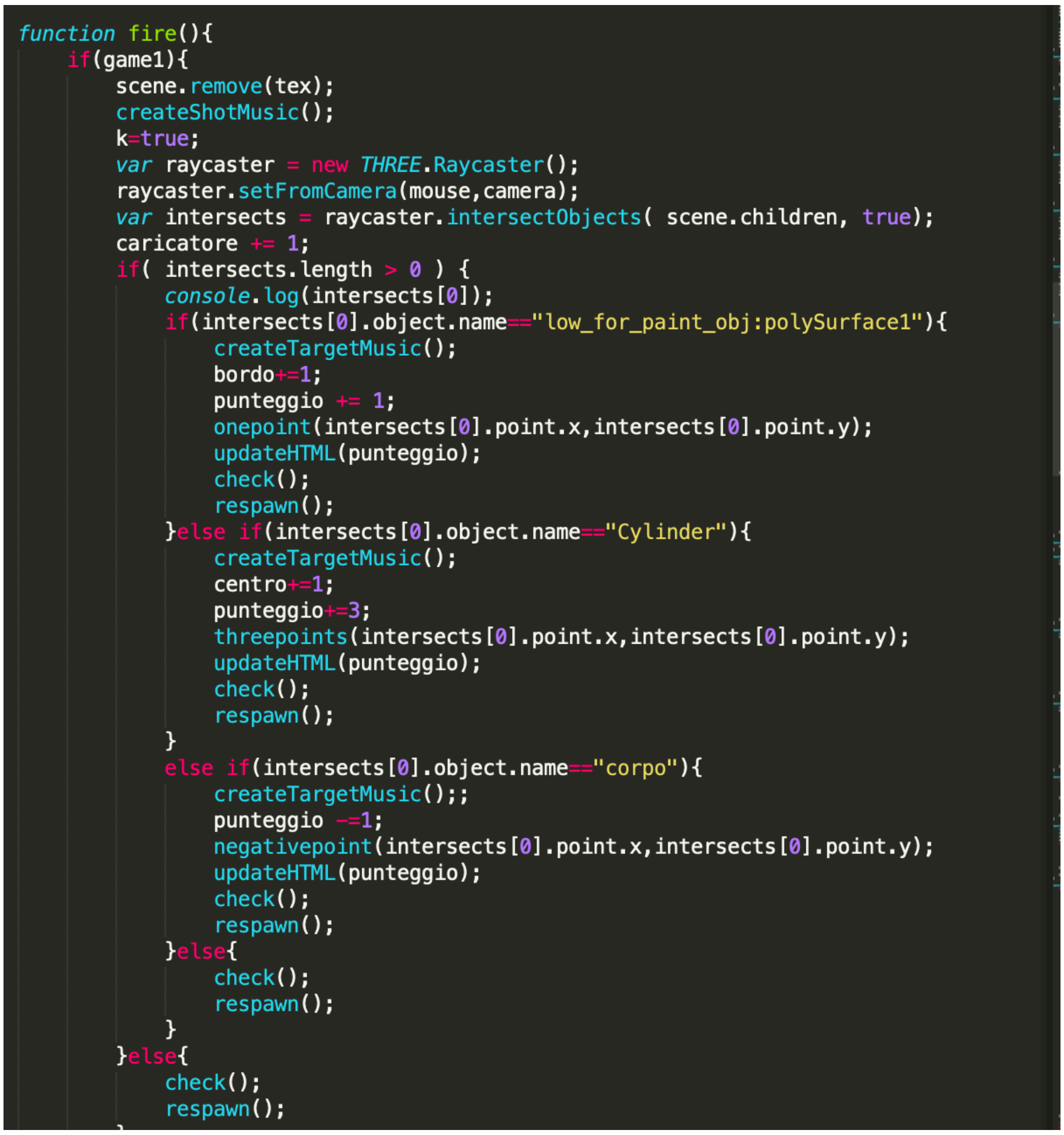


Figure 1: Gun model

The function *fire( ),* has been implemented in order to manage the user hits. When the cursor intersect the right element, in this case the target, we have to make a distinction based on which part of target we was taken.
Through to *intersects[0].object.name,* if the element is equal to *"low_for_paint_obj:polySurface1"*, user's score will increase of one, if it is equal to *"Cylinder"*, user's score will increase of three while if the element hit is equal to *"corpo"*, the score will decrease of one since he hits a duck.
Inside of each *if,* we call the functions *check( )* and *respawn( ).* In order to update the level value and the score value, we have created others two functions: *updateLevel(x)* and *updateHTML(x)* to get the element by ID and write a new variable x on the screen.



```
function fire(){
    if(game1){
        scene.remove(tex);
        createShotMusic();
        k=true;
        var raycaster = new THREE.Raycaster();
        raycaster.setFromCamera(mouse,camera);
        var intersects = raycaster.intersectObjects( scene.children, true);
        caricatore += 1;
        if( intersects.length > 0 ) {
            console.log(intersects[0]);
            if(intersects[0].object.name=="low_for_paint_obj:polySurface1"){
                createTargetMusic();
                bordo+=1;
                punteggio += 1;
                onepoint(intersects[0].point.x,intersects[0].point.y);
                updateHTML(punteggio);
                check();
                respawn();
            }else if(intersects[0].object.name=="Cylinder"){
                createTargetMusic();
                centro+=1;
                punteggio+=3;
                threepoints(intersects[0].point.x,intersects[0].point.y);
                updateHTML(punteggio);
                check();
                respawn();
            }
            else if(intersects[0].object.name=="corpo"){
                createTargetMusic();;
                punteggio -=1;
                negativepoint(intersects[0].point.x,intersects[0].point.y);
                updateHTML(punteggio);
                check();
                respawn();
            }else{
                check();
                respawn();
            }
        }else{
            check();
            respawn();
```

Thanks to *createTree(x, y, z)* and *createBird(x, y, z)*, we could add to scene several of these model, understood as obstacles, in order to increase the difficulty of the game for each level.
Through to *animateBird( )*, we managed the animation of the duck from the alternation of the legs to the ability to go back when it has crossed the entire space available, making a rotation on itself.

In order to create a target, we have created *createTarget(x, y, z).*
The target is an external model that was implemented with our model in order to obtain a *cylinder* to have an area thanks to which we can distinguish the user hits in the right way.
The *createMovingTarget(x, y, z)* function, is a function used to give to object the direction in order to move it. These two functions are called inside the function *respawn( ).* Thanks to *respawn( ),* we have the possibility to create, after that the target has been hit, a new target where *the parameters* of createTarget and createMovingTarget are generated by Math.random.

In function *check( )*, we have planned all we need in order to manage the levels. The way adopted was to use a precise number of possible shots for each level, using the var *caricatore*. At the end of a given number of shots, we can calculate the accuracy given by [number of target hit/number of available shots]. If this value will be major than 0.5, when I will see a window where I can read all of my stats, I can decide to go to the next level, otherwise I have to return home. The same approach will be adopt for each level. We have to remember that the accuracy result is calculated considering the sum of each level, starting from the first, so a bad result in the first level, even if gives you the possibility to go on, it may not be enough to win the game. Obviously if you don't achieve an accuracy in order to go on, you have only the possibility to restart.

onMouseDown(evt) and onMouseMove( ) are two functions and the first one is used to call the function *fire( )*, while the second function is used to manage the gun position based on the cursor direction.

Using *onepoint(x, y), threepoints(x, y) and Negativepoint(x, y)* based on which part of target the user hits, we have used THREE.FontLoader( ) and THREE.TextGeometry( ) in order to show a writing that will have a position depending on where the target was hit.
If the user hits the *cylinder* (the red part), he will have 3 points, instead if the user hits the white part of the target he will have 1 point, if the user hits the duck will have -1 point.

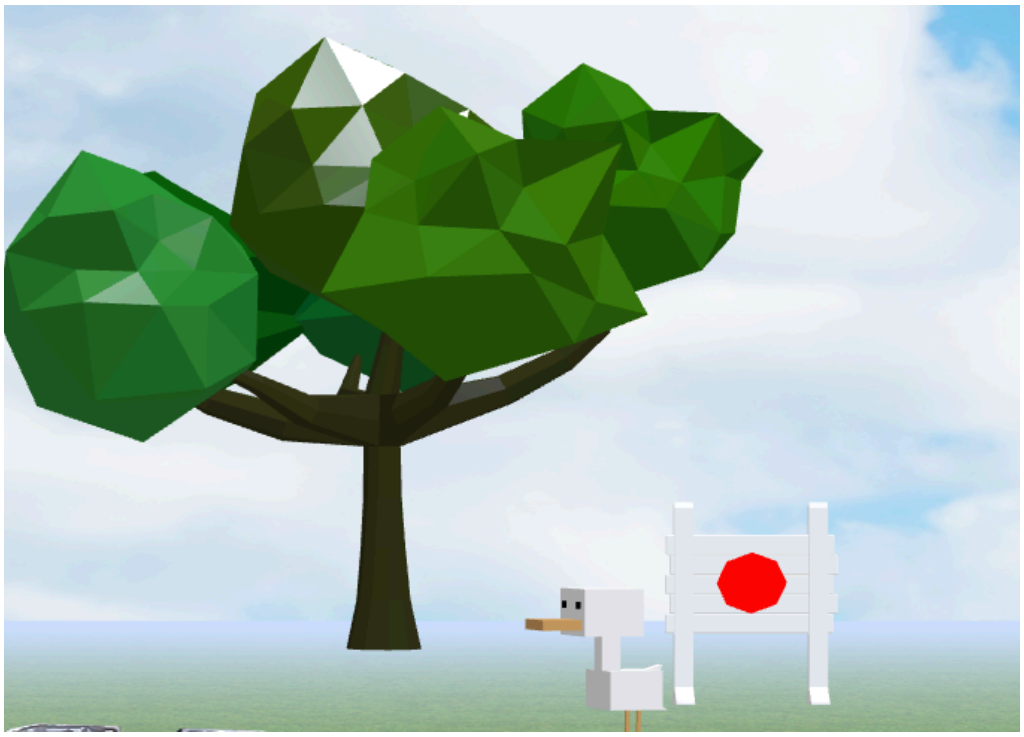
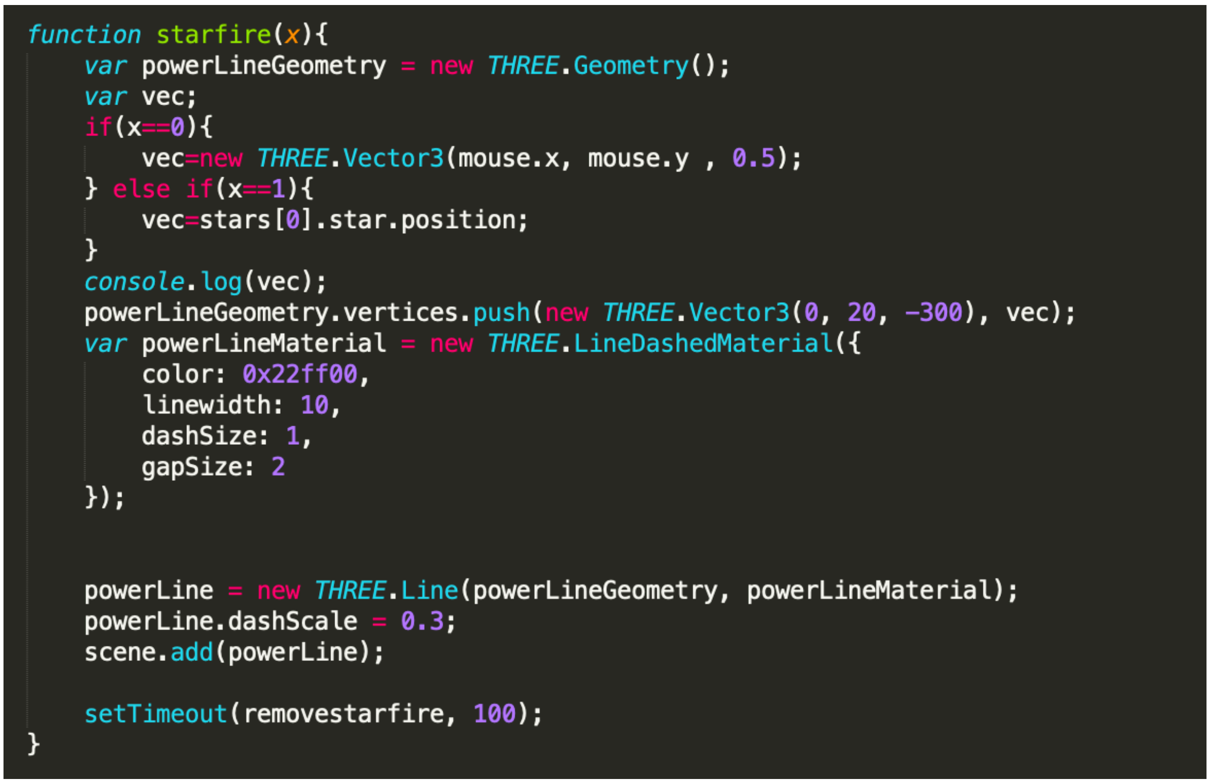


Figure 2: Tree, duck and target model

4. Space Gun

In this scene we don't have levels to overcome but, even if the structure is equal to the first game, in this case we find a different gun model, the presence of some planets during the game that works as obstacles and a star to hit. The goal is to achieve a perfect accuracy.

In order to realise this scene, we have used first of all the *createStarGun( )* function to create the right model that will be used by the user in first person with the same interactivity of the first scene.
Whenever the user clicks on the mouse, enable the function *fire( ),* thanks to which we could update the user's score if the star was it. Playing, we can notice that for each shoot that goes to target, appears a line implemented with *starfire(x)* where, using THREE.Geometry( ) and THREE.LineDashedMaterial( ), we have created a *powerLine = new THREE.Line(powerLineGeometry, powerLineMaterial);*. The line appears and will be eliminated from the scene thanks to *removestarfire( ).*


```
function starfire(x){
    var powerLineGeometry = new THREE.Geometry();
    var vec;
    if(x==0){
        vec=new THREE.Vector3(mouse.x, mouse.y , 0.5);
    } else if(x==1){
        vec=stars[0].star.position;
    }
    console.log(vec);
    powerLineGeometry.vertices.push(new THREE.Vector3(0, 20, -300), vec);
    var powerLineMaterial = new THREE.LineDashedMaterial({
        color: 0x22ff00,
        linewidth: 10,
        dashSize: 1,
        gapSize: 2
    });


    powerLine = new THREE.Line(powerLineGeometry, powerLineMaterial);
    powerLine.dashScale = 0.3;
    scene.add(powerLine);

    setTimeout(removestarfire, 100);
}
```

The star model is created by calling *createStar(x, y),* where the parameters establish the position of the target in the space. Every time that we want to create a new star, we have to pass it the parameters and this is done by Math.random.

Whenever that the user enables the function fire( ), if the target has been hit, *respawnStar( )* function is called in order to show a new star using the function *createStar(x, y).*

With *createPlanets( ),* we have taken a model composed by nine planets and through the function *traverse* we identify the children of the model, that are equivalent to the nine planets, associating each child to a variable that was subsequently managed in *animate( )* function where each planet will have its own animation.

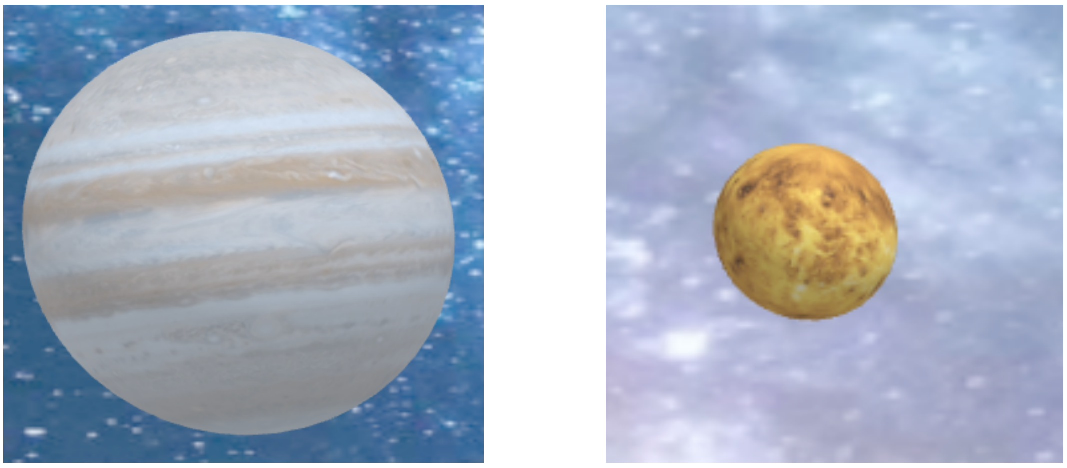


Figure 3: Examples of planets

5. Bowling

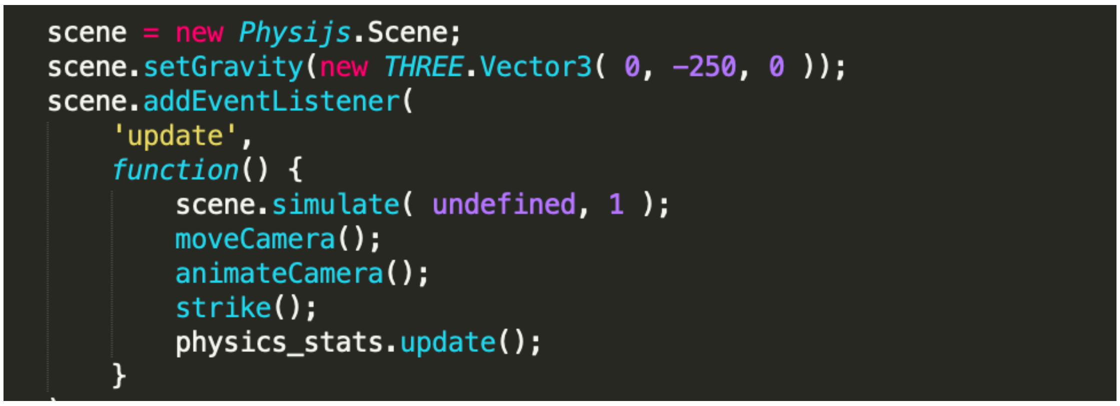
5.1 Physijs: plugin for Three.js

Physijs brings a very easy to use interface to the three.js framework. It makes physics simulations just as easy to run. In fact, there are some steps that must be taken to make a 3D scene come alive.
We need to have the *ammo.js, physi.js, physijs_worker.js* and *three.js* files within our folder structure or coding environment to use Physijs. It uses a web worker to use different threads for physics calculations.
In our case we have used Physijs to develop a third scene in order to realise a simulation of bowling game. The user's goal to give strength to the ball so that the object can fall. As we can see, we have that the camera follows the ball during the path.

5.2 Functions and functionality

In this scene, our goal was to simulate bowling game so the user has to do strike using the ball. The targets effect when the ball collides the pins, was realised thanks to Physijs.

In order to implement the functionality of the game, we have written our code inside of the js script; we have set camera, light (Ambient and Directional) and renderer.



```
scene = new Physijs.Scene;
scene.setGravity(new THREE.Vector3( 0, -250, 0 ));
scene.addEventListener(
    'update',
    function() {
        scene.simulate( undefined, 1 );
        moveCamera();
        animateCamera();
        strike();
        physics_stats.update();
    }
```

Thanks to *physijs.scene* we create the scene and with *setGravity* we are setting the gravity since each object in the physijs scene has to have a gravity value.
scene.AddEventListener gives us the possibility to set a listener in the scene and with *update* we recall the function every time something changes.

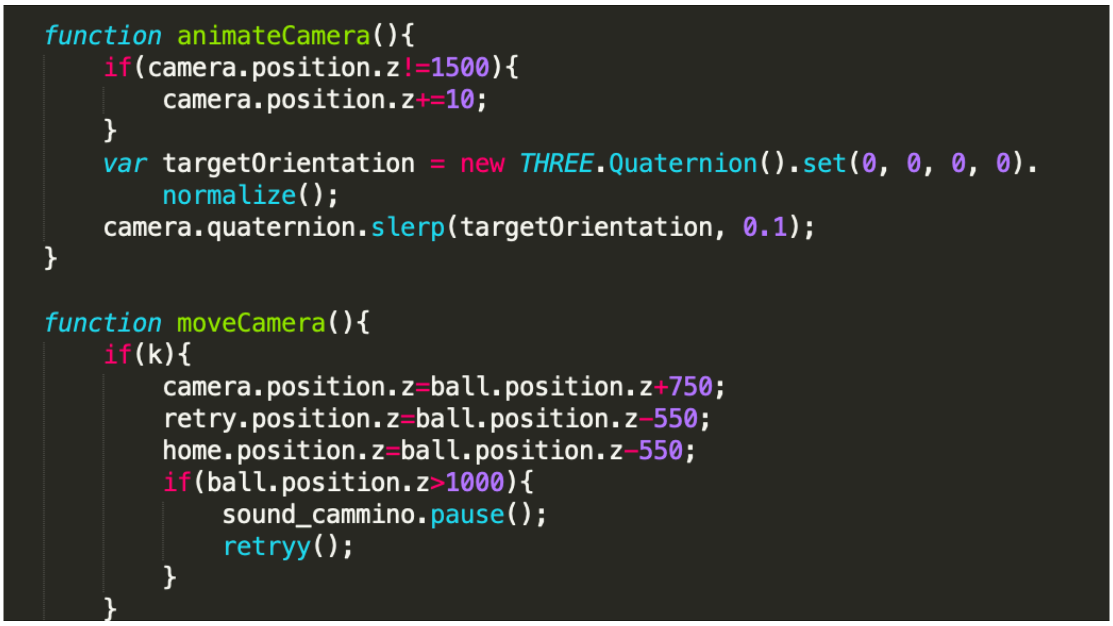
makeBoarder(x, z, w, h) allows us to build our track as a polygon by inserting the measurements to give the correct dimensions to our structure instead with *makeScore*(*x, z, w, h*) we realise a dummy board to make the scene more realistic.

In order to create a pin, that will be our targets, and to set in the right way the position of each of them, we have used *createBoxes( ).*
We had to use addEventListener in order to insert a control collision on the pins when they are hit by the ball.

The item *ball* is created thanks to *SphereMesh*. For all object we have a weight to set. Using *ball.setLinearFactor(new THREE.Vector3(0, 0, 0));* we provide the possibility of movement only on X and Z.
Inside of *createBalls( )* function, we create three different balls, without gravity inside the screen, to provide to user the possibility to choose which ball to play with.

changePowerLine( ) , gives us the ability to see the direction and the power of my shot, based on how much load on the ball is.

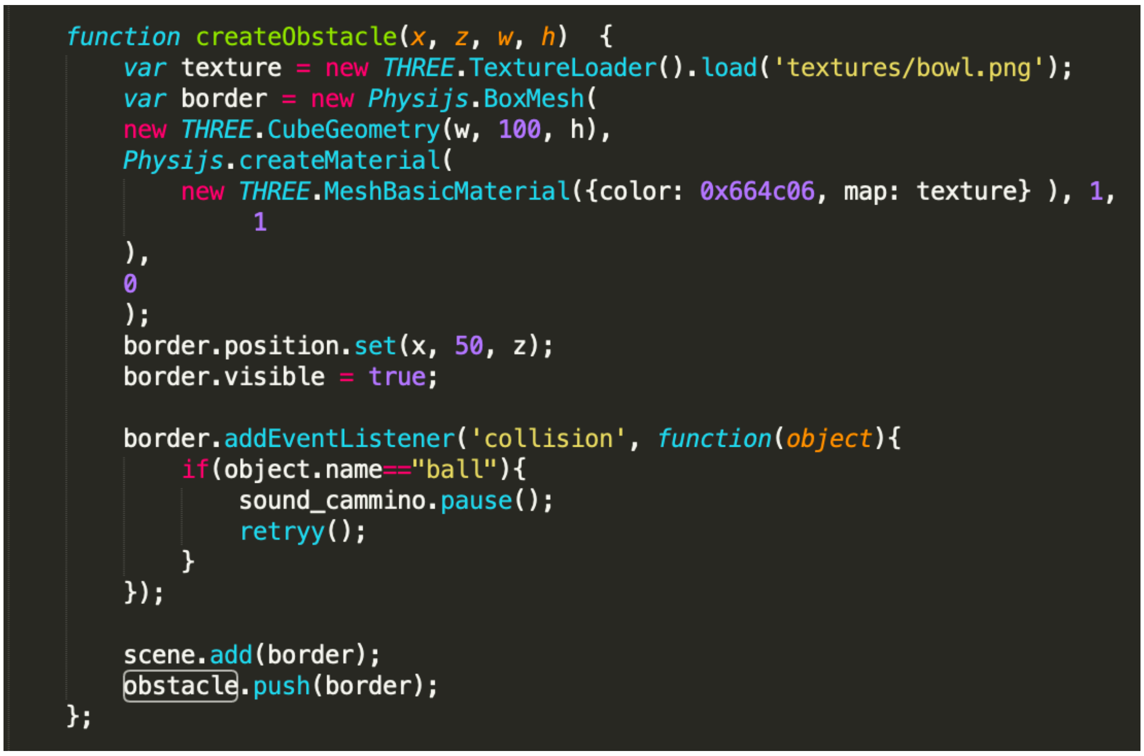
In order to manage the animation of the camera, we have implemented *animateCamera( )* where in the first *if* we move the camera from the pins to the ball and then with *camera.quaternion.slerp(targetOrientation, 0.1)*, we perform the animation. With *moveCamera( )* the camera will follow the ball during is path.


```
function animateCamera(){
    if(camera.position.z!=1500){
        camera.position.z+=10;
    }
    var targetOrientation = new THREE.Quaternion().set(0, 0, 0, 0).
        normalize();
    camera.quaternion.slerp(targetOrientation, 0.1);
}

function moveCamera(){
    if(k){
        camera.position.z=ball.position.z+750;
        retry.position.z=ball.position.z-550;
        home.position.z=ball.position.z-550;
        if(ball.position.z>1000){
            sound_cammino.pause();
            retryy();
        }
    }
```

mouseDownHandler( ), mouseMoveHandler( ) and mouseUpHandler( ) are functions with which we manipulate the ball and the power line. They are the handlers for shooting the ball.

Ritira( ) and home( ) are two functions in order to show the writings on the right part of the screen, which are clickable, in order to retry to play or go to home and choose another game.

Each level that can be chosen directly from the screen, provides the presence of obstacles placed in different positions. Each obtsacle created by *createObastacle(x, z, w, h).*
Whether an obstacle is hit by the ball, the game sequence is immediately restarted and the user is in the initial situation.

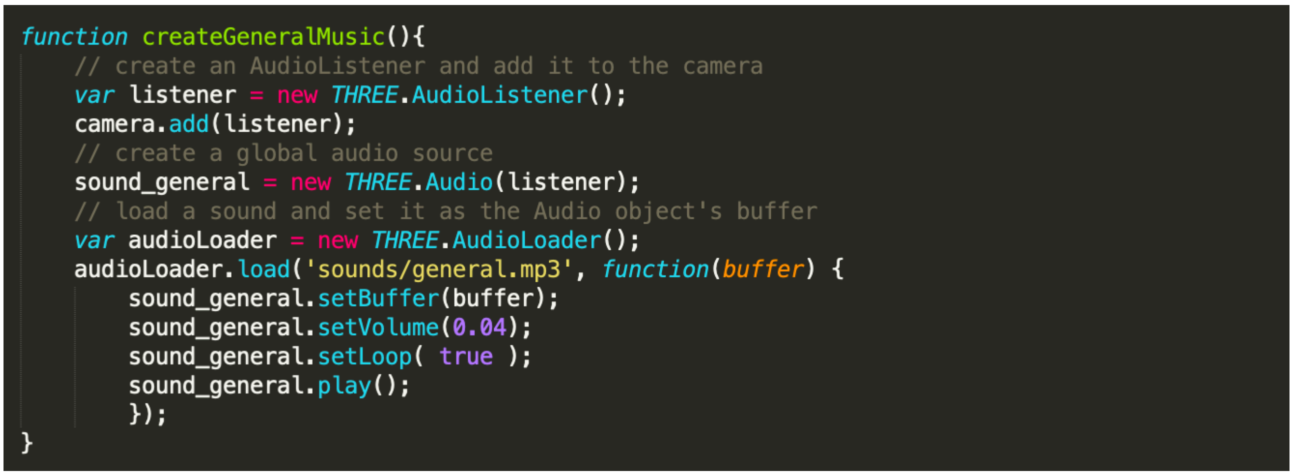


```
function createObstacle(x, z, w, h)  {
    var texture = new THREE.TextureLoader().load('textures/bowl.png');
    var border = new Physijs.BoxMesh(
    new THREE.CubeGeometry(w, 100, h),
    Physijs.createMaterial(
        new THREE.MeshBasicMaterial({color: 0x664c06, map: texture} ), 1,
            1
    ),
    0
    );
    border.position.set(x, 50, z);
    border.visible = true;

    border.addEventListener('collision', function(object){
        if(object.name=="ball"){
            sound_cammino.pause();
            retryy();
        }
    });

    scene.add(border);
    obstacle.push(border);
};
```

6. User's interactions

6.1 Audio

To make the user experience as realistic as possible, we decided to introduce several sounds during the game experience.
Obviously the user can decide to put on pause the general sound by using the control menu and when he wants he can restart it. This has been possible thanks to Three.AudioLoader, we have created an AudioListener and added it to camera. Then we have created a global audio source Three.Audio and finally we have loaded a sound and planned it as the Audio object's buffer.
In order to reproduce sound correctly, we have implemented several different functions for each game modality.



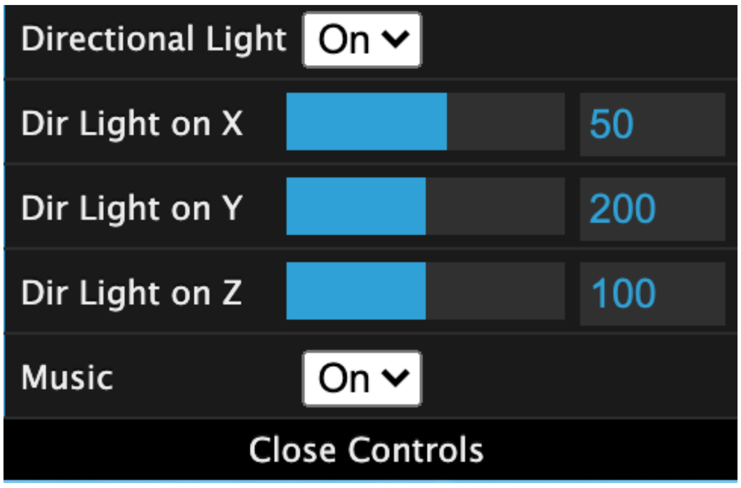
```
function createGeneralMusic(){
    // create an AudioListener and add it to the camera
    var listener = new THREE.AudioListener();
    camera.add(listener);
    // create a global audio source
    sound_general = new THREE.Audio(listener);
    // load a sound and set it as the Audio object's buffer
    var audioLoader = new THREE.AudioLoader();
    audioLoader.load('sounds/general.mp3', function(buffer) {
        sound_general.setBuffer(buffer);
        sound_general.setVolume(0.04);
        sound_general.setLoop( true );
        sound_general.play();
        });
}
```

We have inserted this snipped code to show how we have managed the audio experience. For the general sound, we plan *sound_general.setLoop(true)* to make the playback loop, obviously it starts when I start one of the possible game scenes.

In general we will use a different music function for each effect we want.

6.2 Control menu

In order to manage the effects of directional light on the scene and therefore on the models inside it, we have created what is a menu that can hide itself, thanks to which the user is able to manage in a very simple way the degree of light in according to his tastes. Eventually it is possible to turn off the general music.



Directional Light On
Dir Light on X 50
Dir Light on Y 200
Dir Light on Z 100
Music On

Figure 4: Control menu

The management of this layout is done in the function *initGui( ).*

6.3 Resizing the window

To make the user experience more enjoyable we added event handlers for resizing the window. 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 its properties changed, we need to call *updateProjectionMatrix( )* function.

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

6.4 Commands

To manage the gun or ball and to enjoy the experience, the user can interact with the keyboard and mouse. You can returns to home even if you press *esc*.

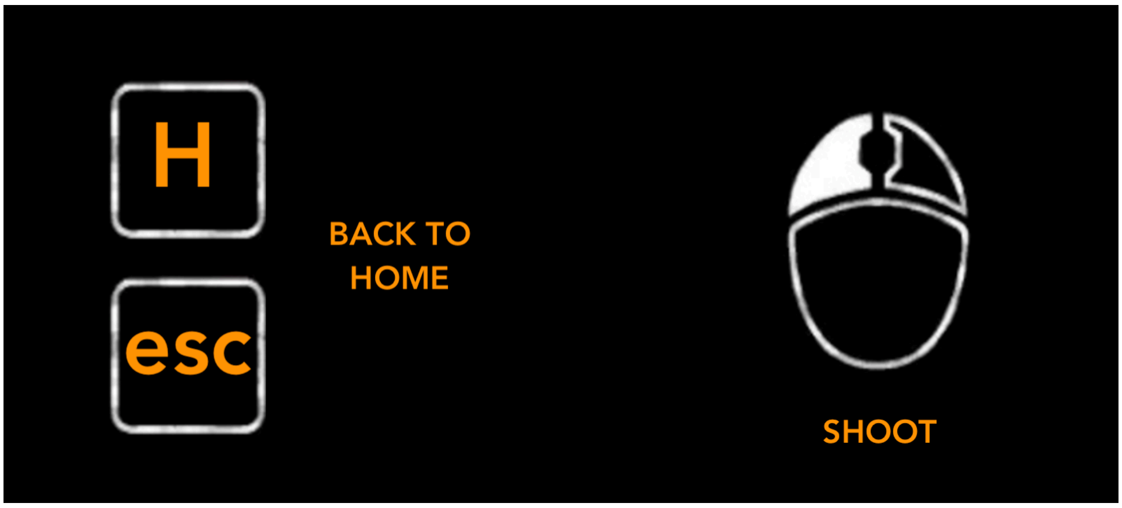


Figure 5: Commands

7. Bibliography

References

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

[2] **Physi.js documentation**
https://chandlerprall.github.io/Physijs/

[3] **Models**
https://sketchfab.com
https://clara.io