Robot WebGL

Realization of a procedural and infinite open world game

Salvatore Cerullo

Firenze, 10/07/2019

INDEX

1 | Introduction

1.1 | Basic Idea

The basic idea is the creation of a game in which a character can move within an **infinite open world**. The world must therefore be generated **procedurally**.

The *two basic concepts* of this project are then:

- **Procedural Generation**: it is a method to create a data algorithmically;
- **Open world**: this term means a video game in which the character can move freely within a virtual world.

1.2 | WebGL & Three.js

WebGL is a low-level JavaScript API for creating 3D interactive graphics on the Web. The standard is based on ***OpenGL ES*** and uses the *<canvas>* element of HTML5.

Three.js is a JavaScript library that provides a WebGL abstraction allowing us to create simple elements in a few lines of code that in WebGL would require a large amount of code.

To implement this application I used the abstraction provided by Three.js and, in some cases, I used ***custom shaders*** written in OpenGL.

2 | THE WORLD

2.1 BlockFloor

2.2 Infinite World

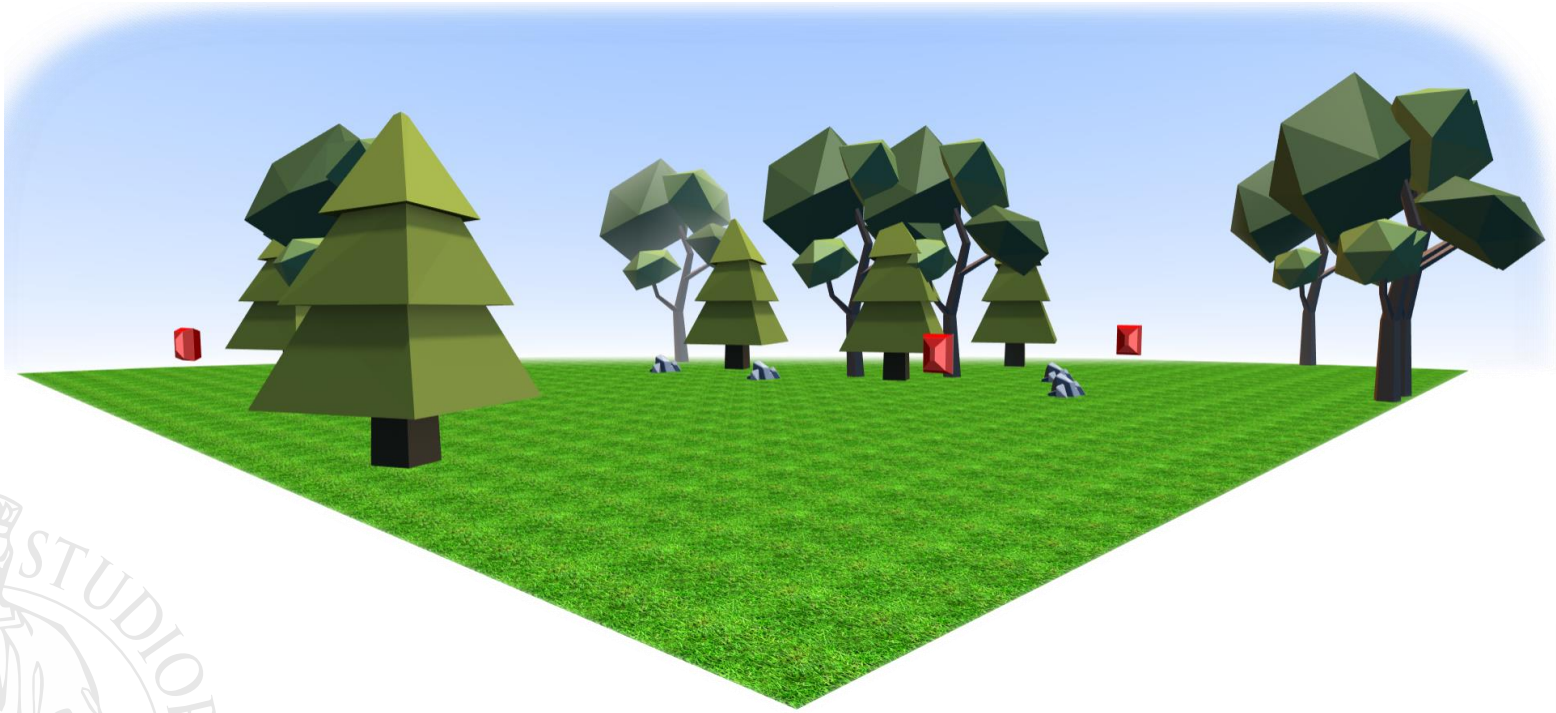
2.3 Crystals

2.4 Trees and Stones

2.5 Sky Dome

2.1 | BlockFloor

The basic element of our world is called **BlockFloor**. This consists of a square plane with dimensions that can be fixed within the *terrain.js* file with the *DIMENSION* variable.
Each block will be constituted not only by the plane, which represents the world terrain, but also by a defined number of objects: *trees*, *stones* and *crystals.*



The single block is managed in the *terrain.js* file where the **BlockFloor** class is defined. For this object, in addition to defining the **constructor**, other methods are also defined that allow the management of the elements that compose it:

- **setPosition(posx, posy, posz)** : set the position of the mash in the position dictated by the posx posy and posz;
- **addToScen()** : adds the mash to the scene;
- **addObjects()** : inserts the trees and stones into the block and scene;
- **addCrystals()** : inserts crystal objects into the block and into the scene ;
- **removeBlock()** : removes all elements of the block from the scene;
- **setTexture()** : inserts the soil texture represented by the block .

The following code is a portion of the definition of the **BlockFloor** class defined in the *terrain.js* file:

```
 9  var DIMENSION = 100; //Indicates the size of a single block.
10  var TREES1_PER_BLOCK = 10;
11  var TREES2_PER_BLOCK = 10;
12  var ROCKS_PER_BLOCK = 7;
13  var CRYSTALS_PER_BLOCK = 5;
14
15  class BlockFloor {
16      constructor(posx, posy, posz, type) {
17          this.dimension = DIMENSION;
18          this.posx = posx;
19          this.posy = posy;
20          this.posz = posz;
21          this.type = type;
22          this.treesCollidable1 = [];
23          this.treesCollidable2 = [];
24          this.rocksCollidable = [];
25          this.crystalsCollidable = [];
26
27          this.setTexture()
28          this.setPosition(posx, posy, posz); //Set the position.
29          this.addObjects(); //I add the trees to the block.
30          this.addCrystals(); //I add the crystals to the block.
31          this.addToScen(); //I add the block made to the scene.
32      }
33      setPosition(posx, posy, posz) {···
36      }
37      addToScen() {···
41      }
42      addObjects() {···
124     }
125     addCrystals() {···
141     }
142     removeBlock() {···
160     }
161     setTexture() {···
175     }
176 }
```

2.2 | Infinite World

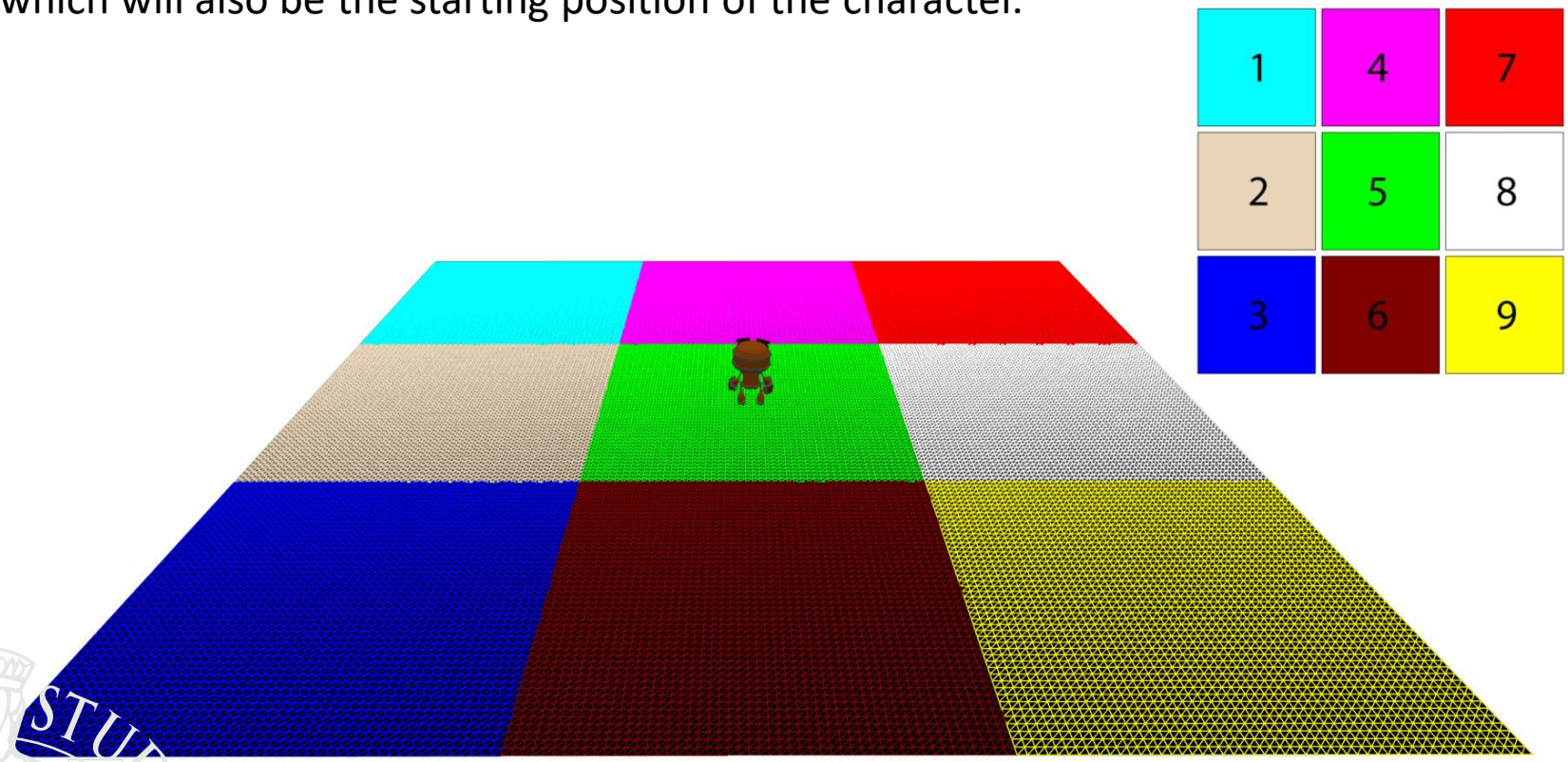
For the realization of the infinite world the program follows the three basic steps below:

1. **Initialization of the world**;
2. **Check of the character's position**;
3. **Updating the world**.

In the following slides these three steps will be briefly explained.

1. Initialization of the infinite world

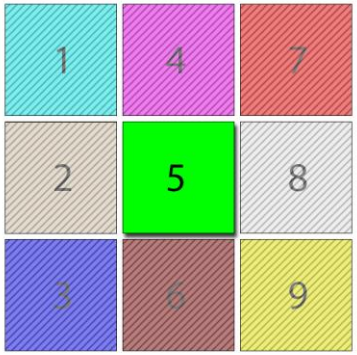
The first phase for the realization of the infinite world is to create a **3x3 grid** made up of *9 BlockFloors*. The *center* of this grid will coincide with the position *(0,0,0)* which will also be the starting position of the character.



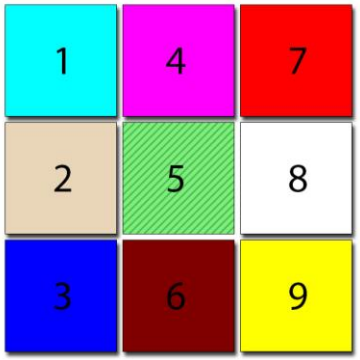
2. CheckPosition

The second operation is a **periodic check** of the character's **position**. During this phase we can have two *cases*:

- **The character is in BlockFloor 5**: in this case I do not make operations on the grid.



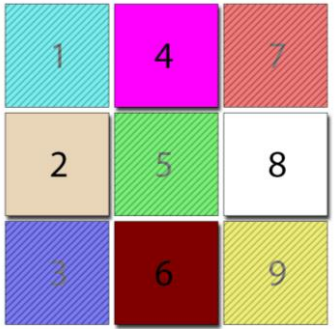
- **The character is not in BlockFloor 5**: I update the grid.



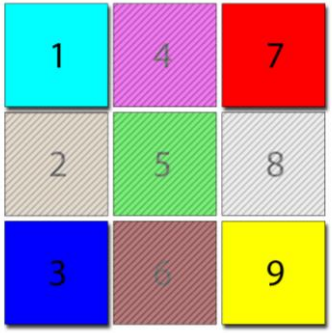
3. Update World

The phase for updating the grid is the most important for the generation of the infinite world. In this phase the application **reorganizes** the world by **eliminating** some blocks and **creating** others.
There are two types of world updates:

- Character in a **border** of the grid if the character enters blocks 2, 4, 6 and 8



- Character in a **corner** of the grid: if the character enters blocks 1, 3, 7 and 9



1. Case Border

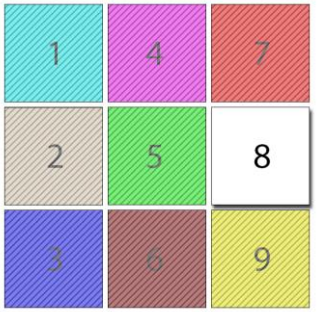


Fig 1.1

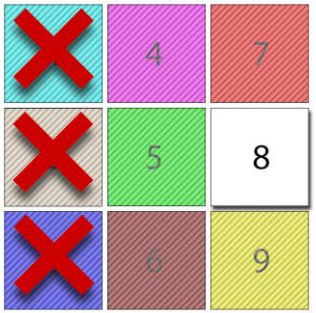


Fig 1.2

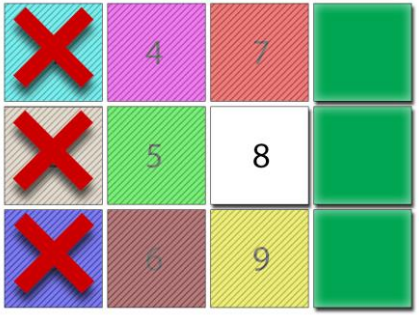


Fig 1.3

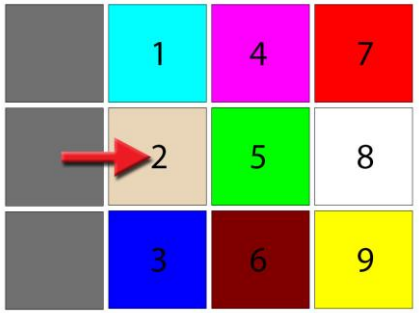


Fig 1.4

2. Case Corner

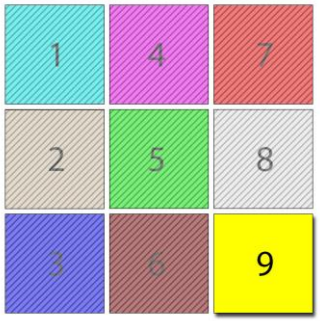


Fig 2.1

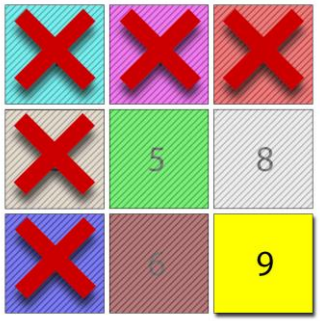


Fig 2.2

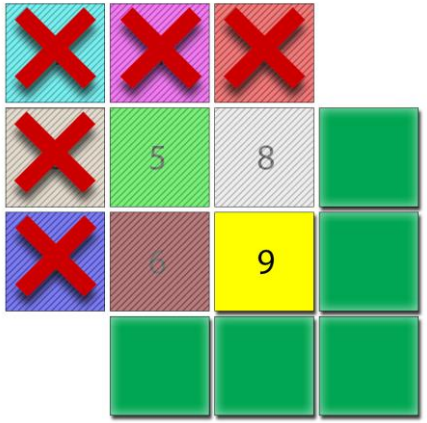


Fig 2.3

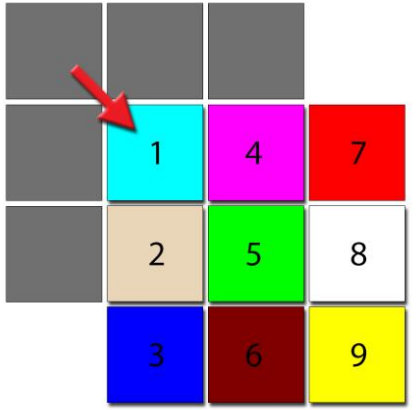


Fig 2.4

2.3 | Crystals

Crystals are 3D objects of the world that can be acquired by the character. These consist of two mash:

- **The internal mash**: is realized thanks to some features made available by Three.js.
- **The external mash**: has the task of realizing a light effect and are made using custom shaders.

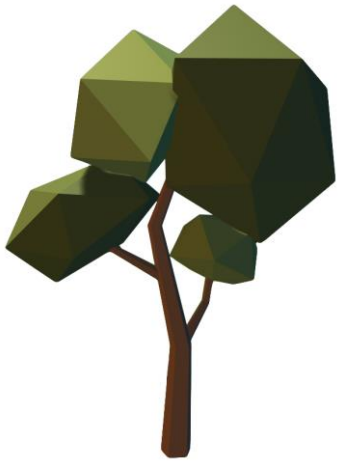
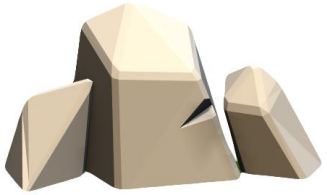
```
<!-- VERTEX SHADER CRYSTAL -->
<script type="x-shader/x-vertex" id="vertexShaderCrystal">
    uniform vec3 viewVector;
    varying float intensity;
    void main() {
        gl_Position = projectionMatrix * viewMatrix * modelMatrix * vec4( position, 1.0 );
        vec3 actual_normal = vec3(modelMatrix * vec4(normal, 0.0));
        intensity = pow( dot(normalize(viewVector), actual_normal), 5.0 );
    }
</script>

<!-- FRAGMENT SHADER CRYSTAL -->
<script type="x-shader/x-fragment" id="fragmentShaderCrystal">
    varying float intensity;
    void main() {
        vec3 glow = vec3(1, 0.5, 0.5) * intensity;
        gl_FragColor = vec4( glow, 1.0 );
    }
</script>
```

The **vertex shader** calculates the dot product of the view vector and the normal, then raises it to a factor (5 in my case) that determines the relative intensity and sharpness of the glow. Higher powers are less intense since the dot product is always less than 1. For calculating the dot product the enhancement I made to the original shader is to calculate the *actual_normal* by multiplying the *modelMatrix* times the normal. That way, even if the geometry is rotated the dot product calculation will work as expected.

The **fragment shader** multiplies the color by the intensity you just calculated. That's all it takes for this "good-enough" glow shader.

2.4 | Trees and Stones

The other objects that make up the world are trees and stones. In particular we have the following objects:

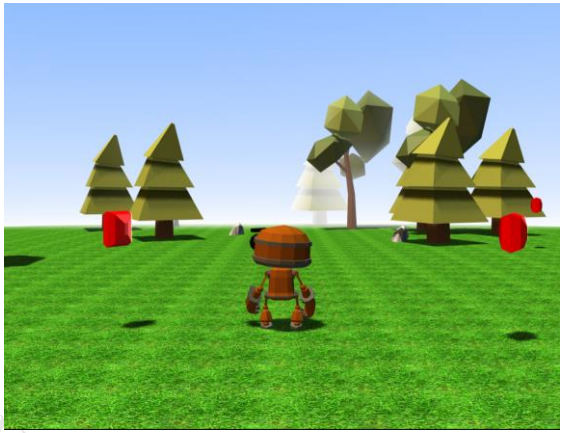
Note: these objects have been downloaded from https://poly.google.com/ and have been imported into my project thanks to the loaders made available by Three.js . These two loaders are:

- **MTTLLoader.js**: to load a material file (*.mtl*);
- **OBJLoader.js**: to load a model file *(.obj*).

2.5 | Sky Dome

The last fundamental element of our world is the **sky**. To simulate the sky I made an object whose *geometry* is **spherical**, while for the *material* I used a second **personal shader**.

The daynight.js file contains all the code to create the sky and to manage the lights. Two environment modes have been implemented: the **day mode** and the **night mode**.



```
74      <!-- VERTEX SHADER SKY -->
75      <script type="x-shader/x-vertex" id="vertexShaderSky">
76          varying vec3 vWorldPosition;
77
78          void main() {
79              vec4 worldPosition = modelMatrix * vec4( position, 1.0 );
80              vWorldPosition = worldPosition.xyz;
81              gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
82          }
83      </script>
84
85      <!-- FRAGMENT SHADER SKY -->
86      <script type="x-shader/x-fragment" id="fragmentShaderSky">
87          uniform vec3 topColor;
88          uniform vec3 bottomColor;
89          uniform float exponent;
90
91          varying vec3 vWorldPosition;
92
93          void main() {
94              float h = normalize( vWorldPosition ).y;
95              gl_FragColor = vec4( mix( bottomColor, topColor, max( pow( max( h, 0.0 ), exponent ), 0.0 ) ), 1.0 );
96          }
97      </script>
```

3 | Collisions

As in the reality, solid objects cannot cross other solid objects. **Collision detection** is therefore necessary to ensure that any area of space cannot be occupied by more than one object.

Another element of fundamental importance is the **resources** employed to carry out the various collision checks with all the elements that make up the world. To eliminate this problem, *the system will verify the collisions with the only elements of the central block that is always occupied by our character*.

In my case I adopted two approaches to identify collisions with an object:

- **Box3**: used to determine collisions with crystals.
- **Raycaster**: used to determine collisions with stones and trees in the world.

3.1 | Crystal Collisions

When a crystal is hit by the character it must be eliminated from the scene and the world through an animation. To do this we must be able to identify a collision between the two objects.
In this case I used an object provided by three.js called **Box3** which is nothing but a cube but which has particular methods such as the ***intersectsBox(Box3)*** which returns a *Boolean* based on an intersection between two cubes.

3.2 | Object Collisions

When the character goes against a fixed object like a stone or a tree, I want it to stay at a certain position simulating reality.
For this type of collisions I used another element called **Raycaster** wich i allows me to apply the raycasting technique.

With the **Raycaster** object I can fire a **beam** in a certain direction by identifying the elements intersected in its path. If the distance to which an object is located is quite far away then I do not perform operations, otherwise I use the operations that simulate the arrest of the character.

To make the character's movement even more realistic, I create a loop of rays that will strike in different directions along a 180 ° angle.

```
*/
function checkCollisionTree(delta, velocity, collidableObjects, minDistance) {
    //Movement Forward (W)
    if (moveForward) {
        var i = 0;
        for (i = -1; i <= 1; i = i + 0.1) {
            //I realize the vectors to which the ray will point.
            var vectorDirection = new THREE.Vector3(i, 0, 1);
            vectorDirection.applyQuaternion(robot.quaternion);
            var ray = new THREE.Raycaster(robot.position, vectorDirection); //I realize
            var intersects = ray.intersectObjects(collidableObjects, true); //I get the
            if (intersects.length > 0 && intersects[0].distance < minDistance) {
                // If the radius returns an intersection with a minor object of minDist
                robot.translateX(velocity.x * delta);
                robot.translateZ(velocity.z * delta);
            }
        }
    }

    //Movement Backword (S)
    if (moveBackward) {
        var i = 0;
        for (i = -1; i <= 1; i = i + 0.1) {
            //I realize the vectors to which the ray will point.
            var vectorDirection = new THREE.Vector3(i, 0, -1);
            vectorDirection.applyQuaternion(robot.quaternion);
            var ray = new THREE.Raycaster(robot.position, vectorDirection); //I realize
            var intersects = ray.intersectObjects(collidableObjects, true); //I get the
            if (intersects.length > 0 && intersects[0].distance < minDistance - 1) {
                // If the radius returns an intersection with a minor object of minDist
                robot.translateX(velocity.x * delta);
                robot.translateZ(velocity.z * delta);
            }
        }
    }

```

4 | Other Elements

4.1 | Cameras

In order to make the gaming experience more versatile, I added the possibility to change the **camera** at any moment of the game using the **C** button. The cameras are as follows:

1 - First Person Camera

2 - Third Person Camera

3 - Free Camera

4.2 | Sounds

Another element that I decided to insert in the application is **audio**. For this purpose I used the tag ***<audio>*** made available by *Html5*.

The sounds that I included are two:

- **Background sound**: is an audio track that will be played in loop during the application execution;
- **Crystals sound**: this sound will be played whenever a crystal is captured.

Note: at startup the sounds are **muted**, to activate them you need to press the **M** button.

4.3 | Top Bar

The **top bar** is a bar that informs the player of the 4 basic settings that can be modified during the game. From left to right we have:

- **Crystal Counter**: displays how many crystals have been acquired during the game;
- **Camera mode**: indicates which camera we are using;
- **Sounds**: indicates the sound status (Mute or Active);
- **Environment mode**: indicates if we are in day or night mode.

5 | Conclusions

The application has been tested on different computers and for a very long time and has given **good results**:

N°	Type	Processor	Graphics Card	RAM	FPS
1	Laptop	Intel Core i5-5200 2.20 GHz	Intel HD Graphics 5500	8 GB	28 - 35
2	Laptop	Intel Core i7-8565U 4.6 GHz	Nvidia GeForce MX150 2 GB	16 GB	54 - 60

In the **future** the application could be *improved* in several *aspects*, such as:

- Inserting clouds and stars for the *sky dome*;
- Change the heights of a land using *heightmaps*;
- Inserting new objects;
- Entering *obstacles* and *enemies*.

Thanks for the attention