Final Project
Interactive Graphics

Désirèe Bellan
1863043

Abstract

This project consist in the realization of a small First Person Shooter (FPS) with a first person camera.

1 Introduction

This project consist in the realization of a small First Person Shooter (FPS) with a first person camera, with three different levels of difficulty and three different characters, each one of them having different characteristics.

It takes mainly inspiration from the fantasy fiction culture and ambience, but with a twist : you will take the roll of three fearless cats: a knight, an archer and a wizard.

They are going to fight terrible levitating monsters using their magical abilities, traveling though the various scenarios of this dangerous world.

2 Libraries

The main libraries used are:

- THREE.js : used to generate the 3D objects, menage most of the functions implemented;
- Physi.js : used to build the scene, plays an important role in the animation of the fireballs (explained in detail in section 6);
- PointerLockControl.js : used to lock the pointer when the game is running;
- GLTFLoader.js : used to load the model for the scene;
- jQuery.js : used to control efficiently the loading fade;

3 User Choices

The second main choice lead to the user is the level of difficulty they want to play. Each level, from easy to medium has a different scene and different enemies to fight:

- level 1 : Forest, easy mode
- level 2: Ocean, normal mode
- level 3: Volcano, difficult mode

Directly below the two boxes with levels and characters there are three clickable buttons:

- START! : close the menu page and start the game,
- HELP : open an information slide where all the commands are explained,
- BACK : go back to the title page.

As soon as the start button is pushed, the scene start to load the models, while the user waits in a loading screen.

Also during the scene the menu choices play a role, in fact by pressing the ESC key is possible to pause the scene, while being redirected to a pause scene, where you can either return to the game by pressing the button RESUME or exit and go back to the home with the button BACK.[1] By clicking ESC the function **lockChange()** stops the rendering by setting the variable PAUSE true and, most importantly, stopping the time through the function **clock.stop()** of THREE.js.

Conversely when RESUME is pushed, it's called **resume()** which has the job of restarting the time, using the function **.start()** of the clock.

4 Characters

Even thought the fight system will be the same (as explained in ...) they differ in their stats (as explained in the menu) :

- knight: life=50000 strength: 50

[1] This functionality is no longer working.

- archer: life=75000 strength: 25
- wizard: life=100000 strength: 16,7

(a) Knight

(b) Archer

(c) Wizard

Figure 1: Characters

Since the main characters isn't shown in the scene, to improve fluidity of movements and avoid long loading, there is no specific imported model for it, but instead there is a transparent sphere with radius $R = 10$.

The character start at position $(0, 0, 0)$ in the scene, but can be moved freely using either the WASD keys or the directional arrows and the space bar. Every movement of the user is managed by the function **animatePlayer()**, where is also taken care of the limits of the area in which the player can move.

```
function animatePlayer(delta) {
    velocity.x -= velocity.x * 10.0 * delta;
    velocity.z -= velocity.z * 10.0 * delta;
    velocity.y += G * 10.0 * delta;
    if (moveForward) {
        velocity.z -= speed * delta;
    }
    if (moveBackward) {
        velocity.z += speed * delta;
    }
    if (moveLeft) {
        velocity.x -= speed * delta;
    }
    if (moveRight) {
        velocity.x += speed * delta;
    }
    if (!(moveForward || moveBackward ||
        ↪ moveLeft || moveRight) ) {
        velocity.x = 0;
        velocity.z = 0;
    }
    ...
    if (controls.getObject().position.y <
        ↪ 15) {
        velocity.y = 0;
        controls.getObject().position.y =
            ↪ 15;
        canJump = true;
    }
```

By choosing a contained zone of gaming with a much more extended ground surface, the user would have the sensation of playing in an ideally unlimited playground, without having to load a lot of scenographic objects.

To contain all the properties of the character, a dictionary named **cat**:

$$cat : \{model : model, life : life, strength : strength\}$$

The model of the cat is a THREE.Mesh() object with spherical geometry and transparent material, positioned at the beginning of the time in the center of the scene.

Both the life and the strength parameters are defined by the menu choice, and determine the performance of the player during fight, as well as the strategy needed to overcome the enemies.

5 Scene

The three possible scene are quite simple.

They all consist in a plane ground, with different textures, a skybox and a series of 3D object displayed around the field, to recreate different ambiances.

All the models are loaded at the beginning od the scene, during the loading phase, using the GLTFLoader library (as already mentioned). For both the forest scene and the ocean scene the models are uploaded though a map and randomly displaced in a ray of 2000 pixels from the origin (and consequently from the character itself). Instead, for the volcano scene the models are use to define the perimeter of the fight area.

Also the scene light changes in the three level, taking different colors and shades, while remaining in the same position. There are two light, an ambient light and a directional light.

6 Monster model

The monsters of the Dungeons and Cats world are levitating heads with big shiny eyes and long skinny arms.

The overall model is obtained through simple shapes, a sphere for the head, eyes and pupils and a box for the arms.

Figure 2: Forest tree

(a) forest (b) ocean (c) volcano

Figure 3: Ground textures

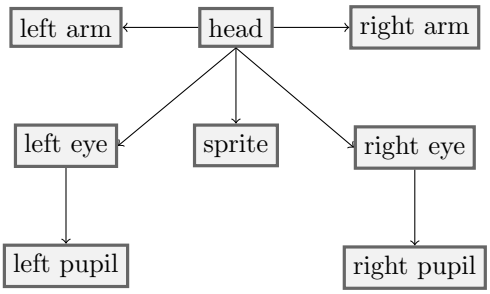

Figure 4: Monster model hierarchy

In the same way as the character, also for the monsters the model has been assigned to a dictionary, as well as other relevant parameters:

```
monster = { model: model, life: life,
    ↪ strength: strength,velocity:
    ↪ monsterVelocity, load:load };
```

The variables **life** and **strength** depends on the difficulty of the level and are determined by a switch condition over :

- Forest: strength=50, life=100
- Ocean: strength=100, life=200
- Volcano: strength=150, life=300

The **velocity** is a **THREE.Vector3()** (i.e. a vector of the three.js library) set at zero when the monster is generated; it is crucial when trying to menage the monsters motion through time, especially for the bouncing and levitating component.

Finally, **load** is a boolean variable set to **true** when the monster is generated.

To collect the monsters and update them collectively, after the generation they are stored in an array **monsters=[]**.

The monsters generation chain start from the function **generateMonsters (clock, monsters.length, ENEMY)**, called in the **animate()** function. It takes as input the clock time, the length od the monsters array, i.e. how many monsters are in the scene and the boolen ENEMY, and returns as output the updated value of ENEMY. This last parameter is used to define the time of generation of the monster wave.

In the function generateMonsters() is called the function **monster()**, the one that actually generate the monsters.

```
function monster2(idx) {
...
var model = new THREE.Mesh(new THREE.
    ↪ SphereGeometry(5, 10, 10),
    ↪ material);
var eyeR = new THREE.Mesh(new THREE.
    ↪ SphereGeometry(2, 10, 10), new
    ↪ THREE.MeshPhongMaterial({
    color: 0xffffff, emissive: 0xffffff,
        ↪  emissiveIntensity: 1,
        ↪ transparent: true, opacity: 1
}));
...
armR.position.set(-2.5, -5, 0);
armL.position.set(2.5, -5, 0);
eyeR.position.set(-1, 0, 3.5);
...
monster.model.add(eyeR);
monster.model.add(eyeL);
...
```

The position of the monster is set so that they are generated in a circle around the camera with increasing angle:

```
monster.model.position.set(cat.model
    ↪ .position.x+150*Math.cos(Math
    ↪ .PI/6*idx), 10, cat.model.
    ↪ position.z+150*Math.sin(Math.
    ↪ PI/6*idx));
```

Finally, the last child of the monster is a **sprite** mesh that act as a health bar positioned just above the top of the ghost's head. Using a sprite is a quite efficient and easy way to have a scalable object always facing the camera.

7 Animations

We can distinguish two big groups of animations: the fireball animation and the monster animation.

7.1 Fireball

The best solution that has been tried to simulate the shooting of the fireball has been using a **Physi.js** scene and declaring the model of the balls as objects with physics. In this way, after setting the position and the direction of the ball, the engine derive automatically the path needed.

To improve the graphic design of the scene, the texture of the balls is set using a fire .png file and the material is set to be emissive.

```
function onMouseDown(event) {
        ...
        raycaster.setFromCamera(mouseCoords,
            ↪  camera);
        ...
        var ballTexture = loader.load('
            ↪ textures/fire.png');
        ...
        var ball = new Physijs.SphereMesh(
            new THREE.SphereGeometry(
                ↪ ballRadius, 10, 10),
            ballMaterial,
            ballMass
        );
        ...
        ball.position.copy(raycaster.ray.
            ↪ direction);
        ball.position.add(raycaster.ray.
            ↪ origin);
        ...
        pos.copy(raycaster.ray.direction);
        pos.multiplyScalar(80);
        balls[balls.length - 1].
            ↪ setLinearVelocity(new THREE.
            ↪ Vector3(pos.x, pos.y, pos.z))
            ↪ ;
        ...
}
```

To get the position and direction of the ball, a ray is casted from the camera using the function **.setFromCamera()**, that takes as arguments the camera of the scene and the coordinates of the mouse controller.

This function is executed every time the left mouse button is clicked.

After having succesfully generated the ball, the interaction with the monsters must be defined. This job is performed by the function **damage()**, that :

1. check if the ball has collided with the monster through the method **THREE.Sphere. intersectsSphere()**;
2. set $monster.life- = cat.strength$ and displays the new value;
3. scale the monster sprite to visualize the life loss.

Figure 5: Fire texture

7.2 Monster animation

The only propose of the enemies is to follow the player and kill him.

The movement is is lead to the function **updateMonsters()**, called after the generating function monster2() in every. It simply compute the direction of the vector with origin in the monster centre and pointing towards the hero, just by subtracting one to the other and then normalizing the result.

```
function updateMonster(delta) {
    for (let i = 0; i < monsters.length; i
        ↪ ++) {
        var speedMonster = 60 * (0.9 + i /
            ↪ 10);
        ...
        direction.subVectors(position, cat.
            ↪ model.position);
        if (35 < direction.length()) {
        //procedes along -direction.
            ↪ normalize()
        ...
        }
        /* else if (20 > direction.length())
            ↪  {
        //back up along the same direction
...
    }
```

$$v1 = cat\ position,$$
$$v2 = monster\ position$$
$$direction = v2 - v1$$

Then the monster move in that direction with a velocity depending on its index in the monsters array. This avoids having all the monster stuck in a queue in front of the camera.

To handle the collisions between monsters, and avoid that they could merge in just one position, inside updateMonsters() is called the function **collisionMonsters()**. It workshop approximately as the function damage() : check the intersection between the model branding spheres and than move each monsters involved in the collision backward along the direction of collision.

Even if the collision is successfully avoided, it causes a source of instability that can lead, in the long time, to unexpected behaviours.

An improvement to the animation, which however caused even more instability, consists in the simulation of the up and down bouncing of the monsters.

```
if (position.y > 12) {
          monsters[i].velocity.y += G * 10
              ↪ * delta;
      }
      else if (position.y <= 10) {
          monsters[i].velocity.y += 10;
```

It works exactly like the movement along the y direction of the camera, but whenever there are many monsters it's summed up with the collisionMonsters() result, causing huge velocities and potential high instability.

This problem has been partially solved by limiting the possible velocities along the y direction and the maximum reachable height.

Finally, to make the monster look at the camera, it's added the line :

```
    monsters[i].model.lookAt(cat.model.
        ↪ position);
```