Project IG

Carrozzo Leonardo Maria, Francione Valerio, Gallo Francesco Giuseppe, Muserra Davide

July 2023

1 Introduction

The aim of the project was to create a survival First-Person-Shooter in which you have to defend against the zombie that tries to attack and kill you. Once you kill the zombie, a more powerful one spawns.

1.1 Libraries used

For the whole project we used only Havok, a physics engine, and BabylonJS, an open-source library built on top of WebGL. The key factors that were fundamentals for us to use it were:

- The easy integration with physics engines (in particular with CannonJS, which is the default one, and Havok, the one which we have chosen).
- The overall performance.
- Possibility to render the scene with a various kind of different materials, textures, such as color, normal, specular, ambient, metallic-roughness and emission.
- Possibility to organize the scene in a hierarchical way.
- Using the glTF loader in order to load complex models.
- Easy to setup lighting and shadows.
- Easy to implement a post-processing motion blur effect.
- Possibility to clone meshes.
- Built-in animation management for complex models, supporting keyframes, easing functions, bones.

1.2 Other tools used

- **Blender**: A free 3D model editor used to view our models and modify them if needed
- **Babylon Sandbox**: A sandbox for 3D model visualization of glTF and other types of 3D models

2 User manual

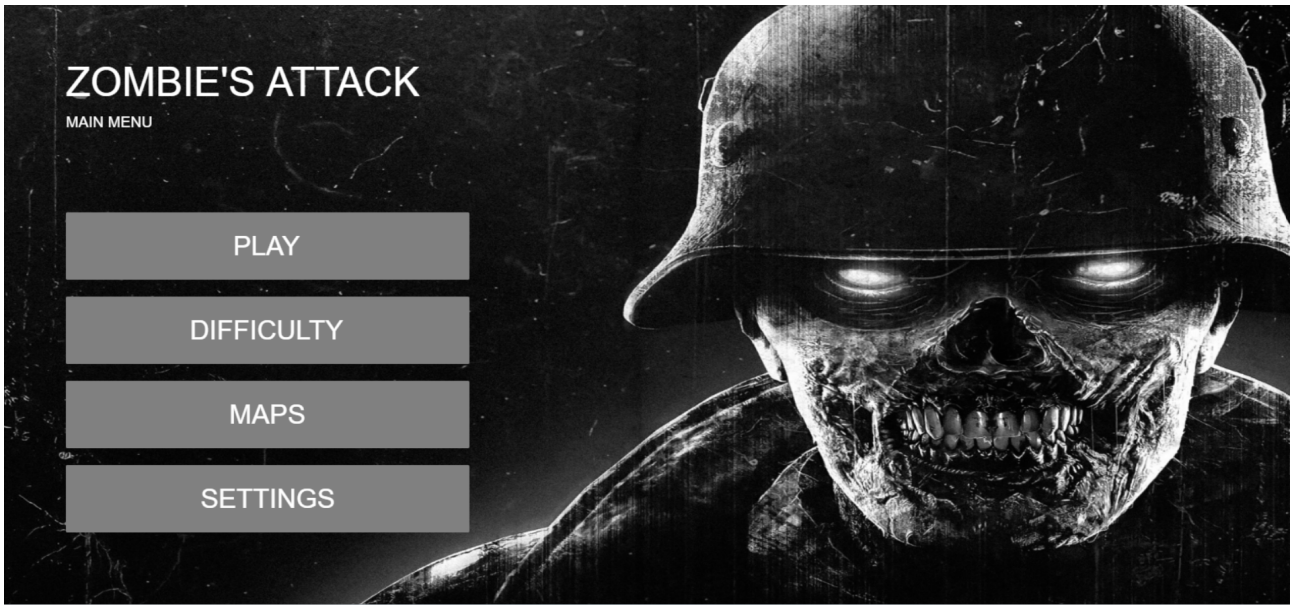
This section describes how the user can interact and use our game.

The core of our game is a survival based FPS in which you have to keep yourself alive as long as possible, while keeping killing the zombie, a non-dead creature that tastes the blood, that is in front of you, ready to damage and kill.

2.1 Main menu

The main menu is a scene realized entirely with the BabylonJS UI module. The user has various actions to take:

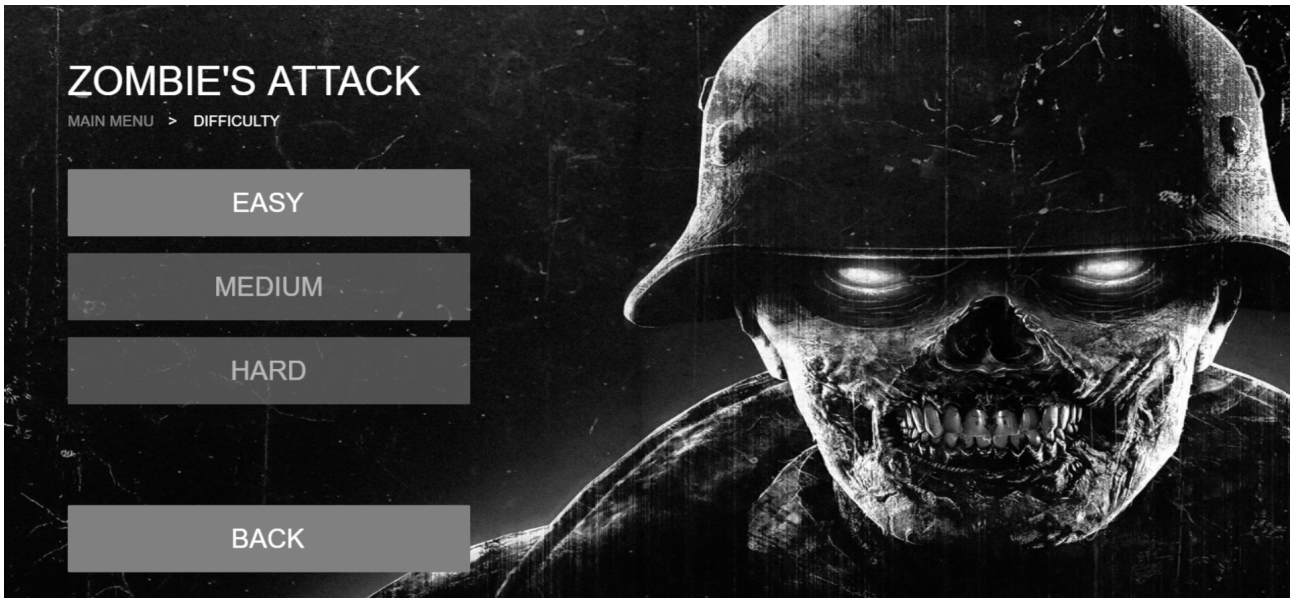
- **PLAY**: By clicking the play button, the game starts with the chosen configurations, such as map, difficulty and many other options.
- **DIFFICULTY**: By clicking the difficulty button, a new scene is rendered. In this new scene the user can choose the difficulty he prefers to play with.
- **MAPS**: By clicking the maps button, a new scene is rendered. In this new scene the user can choose which map he prefers to play in.
- **SETTINGS**: By clicking the settings button, a new scene is rendered. In this new scene the user can choose among various options based on his needs.



2.2 Difficulty menu

The main menu is a scene realized entirely with the BabylonJS UI module. The user has various actions to take:

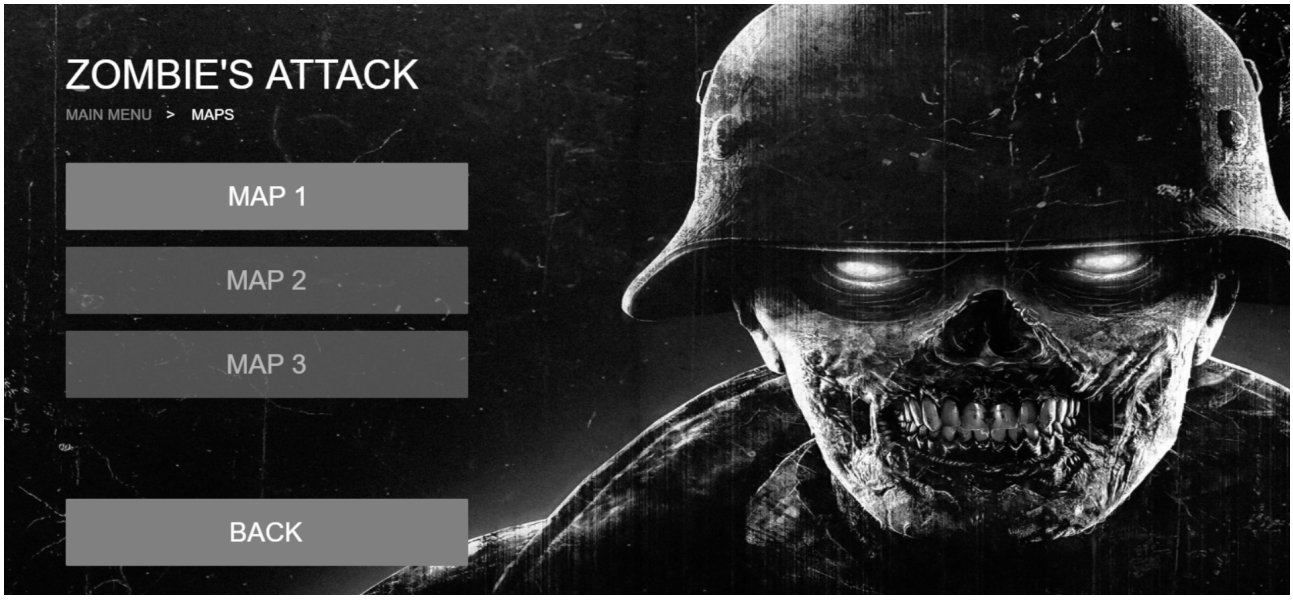
- **EASY**: By clicking the easy button, the difficulty is set to easy.
- **MEDIUM**: By clicking the medium button, the difficulty is set to medium.
- **HARD**: By clicking the hard button, the difficulty is set to hard.



2.3 Maps menu

The main menu is a scene realized entirely with the BabylonJS UI module. The user has various actions to take:

- **MAP 1**: By clicking the map 1 button, the map in which the user plays is set to the first one.
- **MAP 2**: By clicking the map 2 button, the map in which the user plays is set to the second one.
- **MAP 3**: By clicking the map 3 button, the map in which the user plays is set to the third one.



2.4 Settings menu

The main menu is a scene realized entirely with the BabylonJS UI module. The user has various actions to take:

- **MOTION BLUR**: By clicking the motion blur button, it switches to ON/OFF the effect in the game.
- **SHADOWS**: With this option the user has the possibility not only to activate the shadows but also to decides the quality of these.
- **SOUND**: By clicking the sound button, it switches to ON/OFF the volume of the sounds in the game.
- **SOUND - Slider**: By sliding it, the user can choose the sounds volume in the game.
- **SENSIBILITY**: By sliding it, the user can adjust the sensibility of the camera.

3 Technical aspects of the game

An important feature is that the zombie has the possibility to pass through the objects in order to make the game more difficult.

3.1 Controls

The user can control the camera using both WASD and arrow keys. The mouse to shoot to the zombie. As well as other interactions. In particular:

- **W** / **forward arrow**: Allows to move forward in the scene.
- **S** / **backward arrow**: Allows to move backward in the scene.
- **A** / **left arrow**: Allows to move left in the scene.
- **D** / **right arrow**: Allows to move right in the scene.
- **F**: Allows the user to pick up an object.
- **R**: Allows the user to reload the gun.

3.2 Interaction with objects

The user can interact with objects using the F key. In particular:

- **Ammo box**: it picks up ammos, which are used to reload the gun.

3.3 Ray Picking

In order to allow us to make an interaction with the zombie's mesh we used the so-called Ray Picking. It basically creates a ray, from the camera, forward, to the meshes in front of it. The by using a regular expression we check if we have picked the zombie's mesh (actually we pick the hitboxes attached to the zombie's bones).

3.4 Hitboxes

When we have animated our model through BabylonJS, we have seen that our shots were not registered as expected. In particular, the mesh was hit only if shot in the starting position. This is caused because the render of the animation is done on the GPU, while the ray picking is done on the CPU. Since that, we had to attach some hitboxes to the model's bones.

We started by creating an array of objects in which we have listed all the bones that were need for the hitboxes.

We have given them, the bone's name, width, height and depth. Those parameters are passed to the function that creates the hitboxes. Basically the hitboxes are meshes that are invisible. This is done by setting the *isVisible* attribute of the box to false.

Using ray picking and hitboxes we have managed to get the effect we were aiming for.

Since that, we have introduced some values for each hitbox when shot. In particular:

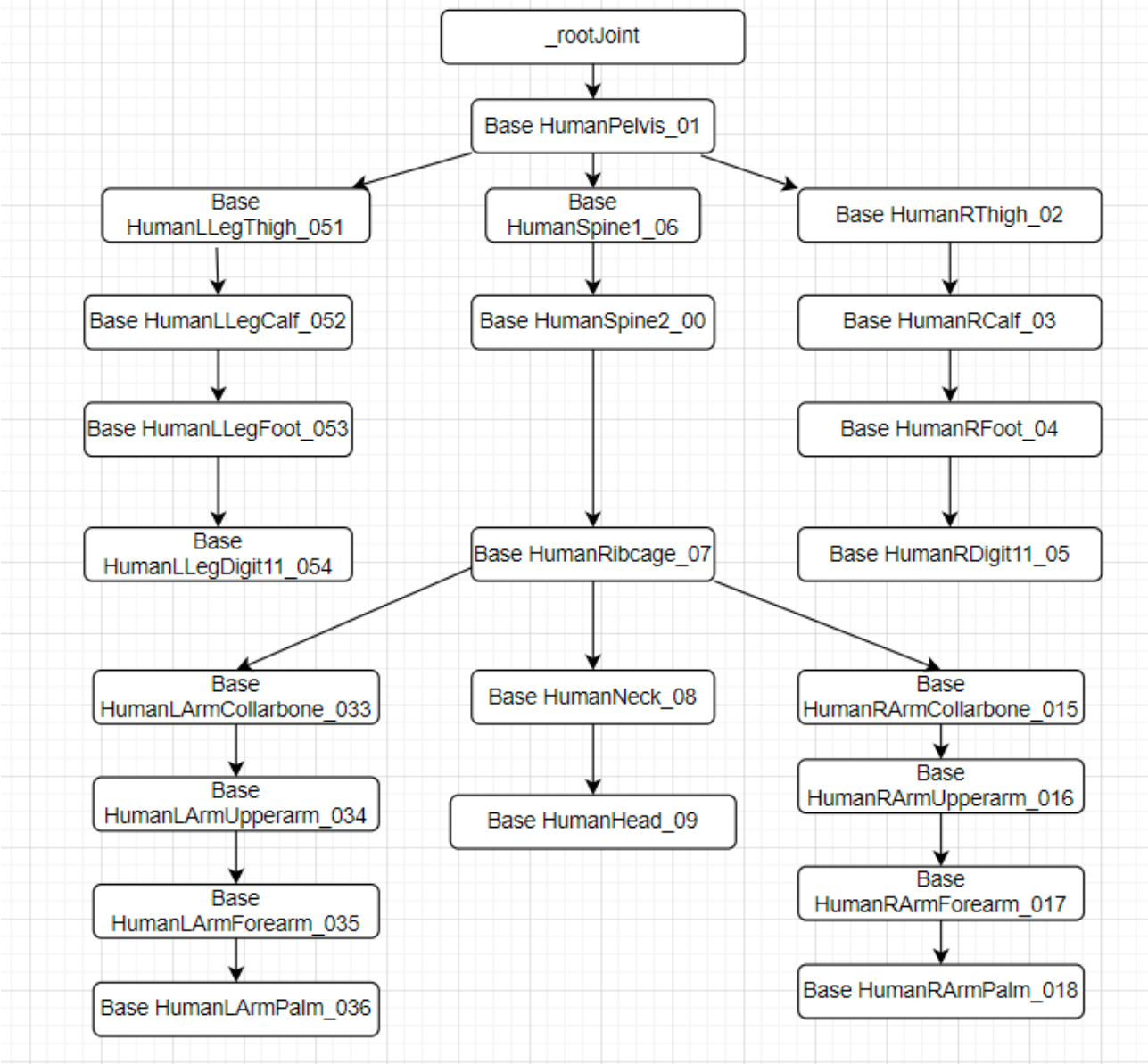
- **Head**: When hit, our gun deals 50 damage to the zombie.
- **Body**: When hit, our gun deals 20 damage to the zombie.
- **Other parts**: When hit, our gun deals 10 damage to the zombie.

4 Models

4.1 Hierarchical models

The enemy and the gun, are a very complex models that includes a hierarchy of bones, which we use to drive the animation of different parts of the mesh.

4.2 Zombie



Zombie is the main enemy of the game. It is composed by several meshes. The animation of the mesh is driven by the hierarchy of bones. They are directly mapped to TransformNode and Bone objects in the Babylon environment. Specifically, this is a list of the relevant bones for animating:

- **_rootJoint** and **Base HumanPelvis_01** are the top nodes from which are connected all the other bones.
- **Base HumanSpine1_06** and **Base HumanSpine2_00** are the bones that allows us to move the torso of the zombie.
- **Base HumanLLegThigh_051** and all the sons are used to move the left leg of the zombie.

- **Base HumanRLegThigh_02** and all the sons are used to move the right leg of the zombie.
- **Base HumanLArmCollarbone_033** and **Base HumanLArmUpperarm_034** are used to move the left shoulder and arm of the zombie.
- **Base HumanRArmCollarbone_015** and **Base HumanRArmUpperarm_016** are used to move the right shoulder and arm of the zombie.

4.3 Other objects

Most other objects in the dungeon are represented by other, simpler glTF models.
Regarding the objects loaded in the maps, they have all been downloaded and discharged from the web. Every model is implemented individually: we import the mesh through an asynchronous function, then we set a position, the scale and the rotation if needed. Then we created a function for the clone of the object.

5 Animation

All animations in the project are realized in the JavaScript code and implemented importing the *BABYLON.Animation* objects, which are part of Babylon's own set of features for animation.

5.1 Zombie animation

In the project all the zombie's animations are implemented by us, we used the keyframes value obtained by manually manipulating the meshes using the Babylon sandbox tool, in order to obtain the right mesh and testing the correct rotation. Although we always worked with Euler angles, in the final animations implemented in the code, we used both quaternions and Euler angles: we used the *BABYLON.Quaternion.FromEulerVector* function to convert from Euler to quaternions. In addition, in order to be able to play and stop the different animation, we created a group for any animation thanks to the *Babylon.AnimationGroup* function. In particular, for the zombie we have three groups of animation, "zombieWalk", "zombieDeath", and "zombieAttack", we alternate this groups in order to adapt the animation of the character according to the situation.

- ZombieWalk: it is the first animation that is activated when the zombie is far, at least of three *, from the camera and it continue in loop since the condition in satisfied. The walk consists in move the legs, arms and the entire body in order that it seems a real zombie walk.
- ZombieDeath: is the animation triggered when the zombie runs out of life, due to the shooting of the first-person character.When we trigger it, it stops the walk animation and start the death. The death animation consists in rotate the body and the legs, so that it looks like it has been hit and then move the height of the mesh so that it looks like it is falling.
- ZombieAttack: this animation is triggered when the zombie is near enough to the camera. At this point we stop the walk animation and start the attack animation. The attack animation does nothing more than rotate the zombie's torso and right arm, in such a way to simulate a simple attack.

In order to render more coherent the animations, as we advance through the game with rounds, we also increase the speed of the animations.

5.2 Ammo animation

Regarding the ammo animations we are going to manage the mesh when we shoot, and in this case what it does is move back and forth along the y-axis. The other animation is the recharging, and in this case we move the ammo down and after few seconds it returns up.

6 Shadows

For the light that create shadows (the *environmental DirectionalLight*, the *Point Light* and the *SpotLight* for the special attacks), the shadow casting is implemented by creating a *ShadowGenerator object* for the light. This object holds the shadow map for the corresponding light source, whose resolution can be decided with a parameter. The *ShadowGenerator* then registers which objects in the scene should produce a shadow because of the associated light via the *addShadowCaster method*; and finally each object that should display the shadows does so, by setting its own *receiveShadows property*.

7 Maps

We created three maps, with three different environments: a cemetery, a city, and a desert. We worked in the same way for all the three maps.
We first created a ground, and we attached to it a texture, different for each map. We implemented a function that imports the *diffuse*, the *normal* and the *AO* texture, and thanks to *PBR material*, we gave them physical features.
Next we started importing several models, in order to populate at best the maps. Each time we needed more than one copy of the same object (for example the trees in the first map), we used the function for the clonation implemented in the models.
Every object is also characterized by a shadow and bounding box, in order to not be crossed by the player.

7.1 MAP1: Cemetery

The idea of map 1 was to have an abandoned graveyard in a spooky environment . By entering the gates of the graveyard we are first faced with some old tombs piled up in the middle of the ground. Behind them a mausoleum stands tall, giving a sacred yet spooky vibe to the scene, and at its sides there are two statutes. All around are crooked tombs and old trees with no leaves. The graveyard is contained by some fences. All the scene is surrounded by a fog. This is the map which contains most models, we have different models for the graves and the trees, while the fences are just clones of the original model put all around the graveyard. All the models have invisible collision boxes which allow to have collisions between the camera and the meshes. Also there are some invisible walls both around the map, so the player can't fall down, and in front of the part of the map with trees and graves, as modeling all the single collision boxes would have been a pretty long and tedious work. As a matter of fact some collision boxes were very fast to create, by just taking the bounding boxes of the meshes, which already given when importing the model, and just creating solid boxes of the same sizes and making them invisible. On the other hand many imported models had pivot points not centered in center of the model but rather in a different point, probably because of the choice of the modeler, so instead of changing the pivot point for each of them, creating an invisible wall came more handy. Regarding the other half of the map, which is the spawn point of the zombie, but also without any models, originally the map had a woods outside of the graveyard, like a small cemetery lost in the

nature, but the map was beginning to become too heavy and the game was not as fluid as we wanted it to be, so ultimately we decided to leave it like this.

7.2 MAP2: City

In a city which is lapsed only your are standing facing the battle for survival. Here basically the same techniques as before were used . Here the central space is bigger in respect to the graveyard, but there are some cars which could become problematic obstacles as your enemy becomes faster and stronger.

7.3 MAP3: Desert

Finally map 3 is the most risky for the player, a lot of obstacles are in his way, which do not seem to be a problem for the enemy. Here the issue was just as map 1, most bounding boxes of the rocks were not centered, so some of the collision boxes are built by hand and placed in the right position.