Interactive Graphics - Final project

Stefano Bonetti - 1764624

1 The Worms series

The following project takes inspiration from a famous series of videogames known as *Worms*, created by Team17 in 1995. Many videogames belonging to this series were released during the years, with the latest (a spin-off) being released in 2020.
Worms is an artillery game, a type of game where two or more teams, generally on a 2D map, battle each other until only one team remains alive. In Worms, the players control a team of small worms, and they have to shoot at the opposing teams, while trying not to hit their own worms. In order to shoot, a player has to take the aim (choosing the vertical angle of the shot), and then decide the power; the more power is given to the projectile, the further it will of course go.
Since projectiles are affected by gravity, artillery games are basically "physics" games. What you have to do is to decide the right angle and power in order to "build" an appropriate parabola for your projectile, so that it avoids possible obstacles and hits the enemies.
Some Worms games were developed in a 3D setting. The core of the game remains the same, and the only real difference is that the gameplay revolves around 3D levels: you can look around and move in different directions, but you still have to aim and shoot with a certain power. One of these 3D chapters is *Worms 4: Mayhem*, which is the game I took inspiration from.

2 Woborms

The gameplay of *Woborms* (from the union of "Worms" and "robots") is, at its core, similar to the one of 3D Worms. Two players play against each other on the same computer alternating their turns, and they both control a team of four robots with small cannons in place of the forearm. During a single turn, each player has a limited amount of time (20 seconds) to move, aim and shoot. In order to play:

- use W, A, S and D to move around;
- press E once to look at the entire map from above, press again to go back to third person view;

- press Q once to see in first person, so that you can take the aim; by pressing again you can exit from first person view;
- when looking in first person, use W, A, S and D again to look around and take the aim (the robot cannot walk around when aiming), then hold the space bar to charge the shot.

When charging the shot, you will see a counter that starts at 0 increase: that number represents how much power was given to the bullet. Of course, a greater power means the projectile will travel further.

The robot will shoot as soon as you release space bar or when the power reaches a maximum of 10. Remember that you only have 20 seconds for shooting, but the countdown will stop right when you start pressing space bar, so that you can charge the shot as much as you like.
After shooting, your robot can't take any more commands, and your shot can either miss, go outside of the map, hit an enemy or hit an ally. In case there are any obstacles that don't allow you to properly see where the bullet goes, you can still use E to check what happens from above (it is anyway advised to use it during the turn to locate enemy robots).

Each robot has three lives, and when they are hit by a bullet they lose one life; a robot with zero lives gets deactivated and can't be used anymore. When a bullet hits something after you've shot or goes outside of the map, the opposing player's turn will start after a small delay.
Remember that bounces are not valid: if for instance a projectile hits a wall and then hits a robot, the shoot will just be considered missed, therefore you need to directly hit a robot in order to damage it. As soon as a player's robots are all deactivated, the game ends and the opposing team wins.

The eight robots are placed in a map with borders, filled with many objects (trees, walls...), so that players can try to hide from the enemies and to make it harder for them to shoot. In general, if you expose yourself it could be easier to aim at the enemies by getting closer, but the same goes for them too; if you decide to hide, you'll be safer but hitting the opponents could be harder.

3 JavaScript libraries

The libraries I used are Three.js (version r130, from which I also took OBJLoader.js to load models in .obj format) for graphics, Tween.js for animations and Cannon.js as physics engine.
A brief description of Cannon.js could be useful for explaining some concepts later.
.
.
.

4 Technical description

4.1 Lights and shadows

The game has two lights: a directional light to simulate the sun, pointing at the centre of the map, and an ambient light to not make the game too dark, both of which are white.
Most objects in the game can also cast and receive shadows (the ground, for instance, doesn't need to cast shadows but only to receive them). The shadows are generated from the light's point of view, and the directional light basically uses an orthographic camera to compute them. Since the camera defines an area in which the shadows are rendered, we need to set it up correctly so that the whole map is included in it, and that's why I changed the camera's `near`, `far`, `bottom`, `top`, `left` and `right` parameters (I used a `THREE.CameraHelper` for this). Unfortunately, increasing these parameters makes the shadows "pixelated", so I increased the shadow map's resolution with `mapSize.width` and `mapSize.height` which made the shadows look significantly better.

4.2 Textures

Most objects in the map have a texture attached to them when creating their material (in general a `THREE.MeshPhongMaterial` to which we have to set some parameters, mainly `color` and `map`). The textures are repeated on the objects an appropriate amount of times horizontally and vertically, and they are all loaded before the game starts.

- The ground has a grass texture as `map`.
- There are three kinds of walls: the walls of the building with an opening, the map's delimitation walls, and some white walls. They all share the same `map` and `normalMap`, but in the building's walls a brown `color` is also added, whereas in the delimitation walls there is grey. The white walls only have the texture and the normal map applied to them.
  Actually, since the faces of a single wall have different dimensions, instead of applying the same textures to all faces I had to specify the textures for each face when creating the wall's material. To do this, I "cut out" from the original wall texture (`wall_color.png`) two thinner textures: one for the side faces of the wall (`wall_color_side.png`) and one for the top (and bottom) face (`wall_color_top.png`). The same goes for the normal map (`wall_norm.png`, `wall_norm_side.png` and `wall_norm_top.png`).
- In the game there are also some cylindrical towers, which have a `map`, `normalMap` and `roughnessMap` applied to them. Since roughness maps are not supported by Phong materials, the turrets' material is a `THREE.MeshStandardMaterial`.
  The top and bottom faces just have a grey `color` instead.

- The barrels' models I found on the Internet came with their own textures, which are simply applied to the imported models as `map`s.
- The trees' trunks are again a simple `map`.

I also thought about handling the textures' *mips* for when they are seen from far away (by setting the textures' `minFilter` property), but I decided not to because no significant difference could be noticed.

4.3 Hierarchical model

The hierarchical model for the robots is the one in figure 1. Ellipses represent `THREE.Object3D`s, rectangles meshes and rounded rectangles cameras.

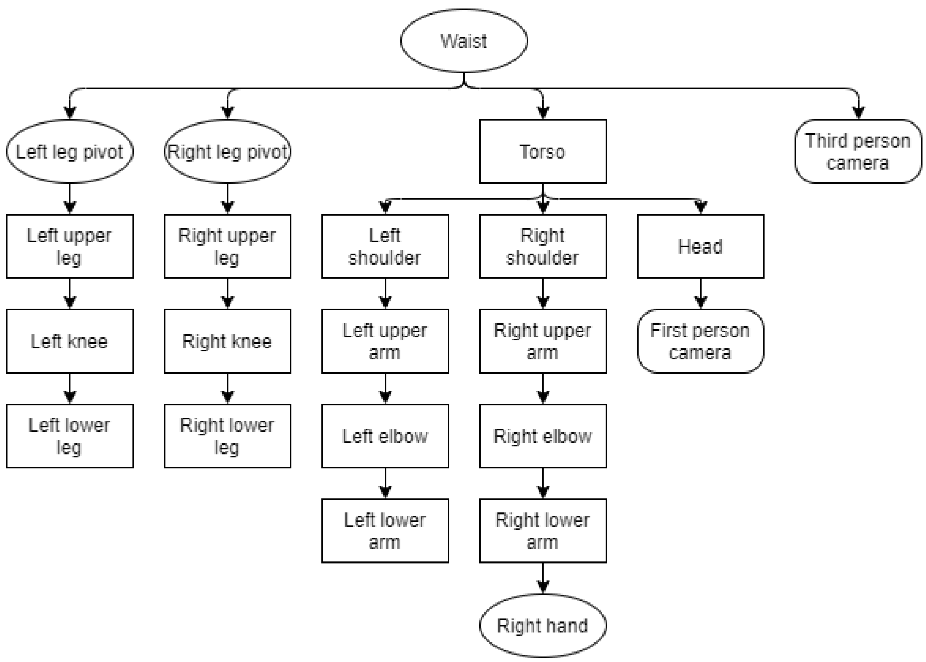

Figure 1: Robot's hierarchical model

The waist "represents" the whole robot, and it's used by the game to know and in case change the robot's coordinates and orientation. The pivots, knees, shoulders and elbows (the last three are all spheres) are used to animate the arms and legs by changing their rotation.
Since some animations require the arms and head to move together with the torso, I decided to make them its "children", so that by moving the torso they all also move accordingly (specifically, the shooting animation makes the torso and arms rotate a little, while the idle animation moves the torso up and down and the arms and head move with it); the legs instead move independently from

the rest of the body and are directly children of the waist.
Each robot has two cameras. The third person camera is the default one and is placed above the robot, slightly behind it; its coordinates and `lookAt` are expressed with respect to the waist. The first person camera is the head's child, so that when the head rotates the camera rotates as well, and is placed inside it looking forward.
All parts of the body are box geometries, aside from the right lower arm which, being a cannon, is a cylinder geometry. The right hand is another `THREE.Object3D`, but it's not used by the animations. When a player is aiming before shooting, the right arm (the one that has the robot's cannon) moves around and points toward where said players is aiming; therefore, when the robot shoots it's important to know exactly where the bullet was shot from. For this reason right hand, being placed slightly ahead of the cannon, is the point where the bullet gets spawned, and when the robot shoots, its right hand's coordinates (with respect to the entire scene) are retrieved and used to make the physical computations for the bullet that was shot.

As for the other elements in the game (ground, walls, towers, barrels and trees), they are all just directly added to the whole scene. This also goes for the global camera looking from above, which points to the centre of the map.

4.4 Animations

The robots can perform many animations.

- Idle animation (in the `robot.js` function `idle`): this looped animation is reproduced when a robot is doing nothing, whether it's the robot who is playing its turn or the other ones that are not being controlled at the moment. When idle, a robot goes from its original position (just standing) to one where its body is slightly lowered and its legs bent; this way, a kind of breathing animations is created, which is not something the robots need to do but I found that perfectly immobile robots were too "unnatural".
  To implement the animation I created two keyframes (as said, one in which the torso is placed in the original position and the legs are straight, one in which the torso is lowered and the legs are bent); to go from one to the other and vice versa, a linear easing function is used.
  Using Tween.js' `chain` function, the first keyframe is chained to the second one and the second one is chained again to the first one, so that the animation is continuously looped.

- Aiming (`toAim`): by changing the right shoulder and elbow's rotation, the robot takes the aim pointing the cannon forward, with a `Quadratic.InOut` easing function.

- Stop aiming (`aimToIdle`): the right arm goes back to its original position with a `Quadratic.Out` easing function, so that the arm descends quickly at first and then slows down. When the animation ends, the idle animation is restarted by calling `idle` thanks to the Tween.js `onComplete` listener.

- Shooting animation (`shoot`): after having taken the aim, the robot shoots. The right shoulder and elbow's angles change because of the recoil, and the torso rotates a little too; we use an `Exponential.Out` easing function to really simulate the power of the recoil.
  After the animation is completed, `aimToIdle` is called to make the robot go back to idle, but this time the animation is started after a little delay with to the `delay` method and also lasts a bit more.

- Walking animation: it is the most complicated one and involves the rotation of shoulders, leg pivots and knees and uses three different functions. In `idleToWalk`, the robot starts walking moving from the original position, bringing the right leg and left arm forward and the left leg and right arm backward. When the animation completes we call `walk`, which uses four keyframes and chains them in a loop. ....
  As soon as the player stops using the walking commands, the walking animation is interrupted and `walkToIdle` is called. Of course, it brings the robot back to the neutral position and then starts the idle animation.

- Hit animation (`hit`): when a robot is hit by a bullet it gets "hurt" and wriggles. At first it quickly (`Quartic.Out` easing) raises its arms and bends its torso back; then by chaining another tween it goes back to the original position (`Quadratic.Out`). As usual, with `onComplete` it goes back to idle.

- Death animation: when a robot is hit and loses its last life, instead of performing the hit animation it just deactivates. I thought about making the robots completely fall on the ground and lay down, but I decided to make them still stand so that the dead robots become new static elements of the map and players can use them to hide behind them.
  The animation is split in two. First, the robot's body slowly descends and its legs bend (linear easing function); then, the robot completely loses control of its body. Indeed, the robot quickly bends its torso and head forward and the arms hang down.
  This last part of the animation is handled with two tweens executed simultaneously. The tween for the head and torso bending forward is chained after the tween for the first part of the animation; by using the `onStart` listener, the tween that handles the arms is started together with it. To reflect the loss of control by the robot, both the last two tweens use a `Bounce.Out` easing function.

5 Elements taken from the Internet

- The barrel model and textures `barrel.obj`, `barrel_0.png` (closed barrel) and `barrel_1.png` (barrel with water) were made by *mastahcez* on opengameart.org;
- `grass.png` was made by *athile* on opengameart.org;
- `stone_wall_color.png`, `stone_wall_normal.png` and `stone_wall_roughness.png` were made by *Katsukagi* on 3dtextures.me; `wall_color.png` and `wall_norm.png` were also made by him on 3dtextures.me;
- `trunk.png` was made by *jamie1* on favpng.com;

6 Description of the single files

6.1 Initial menu

6.2 map.js

6.3 game.js

6.4 robot.js

6.5 utils.js

7 Known issues

Sometimes, the collisions between bullets and elements in the map are not perfect (the bullets bounce in unnatural ways or just go straight through the object). I believe this is a problem given by Cannon.js, because by checking with the Cannon debugger I saw that all the Cannon bodies of all the objects are defined correctly (as you can see in figures 2 and 3). Therefore, I believe that Cannon.js doesn't handle perfectly the collisions involving a fast body (indeed this problem wasn't arising during the first stages of coding the game, when the map was smaller and the bullets were slower).
Anyway, this issue doesn't really negatively affect the gameplay, because what's important to know is what is *the first thing* that a bullet hits, and even when a projectile bounces in an unforeseen way, the game still detects that there was a collision and what bodies were involved. Thus, the game will always correctly know if a bullet missed or hit a robot, the issue is just that it might bounce in a strange way.

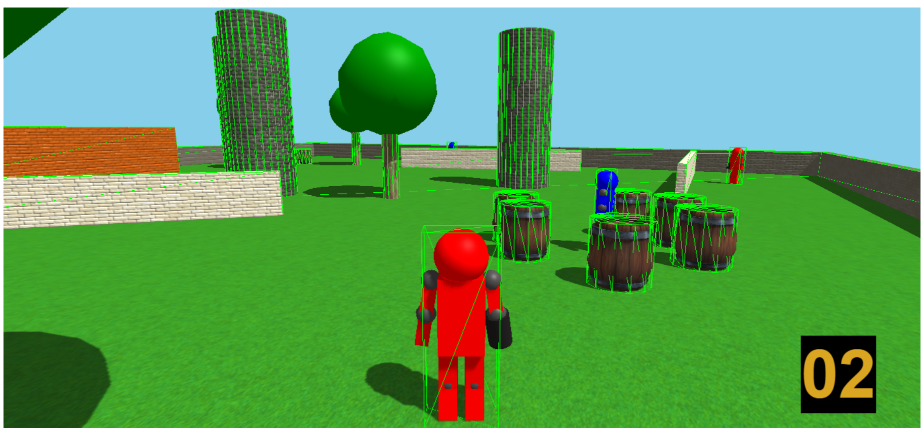

Figure 2: The game with THREE.CannonDebugRenderer activated. You can see that the Cannon bodies (basically the hitboxes) are defined correctly

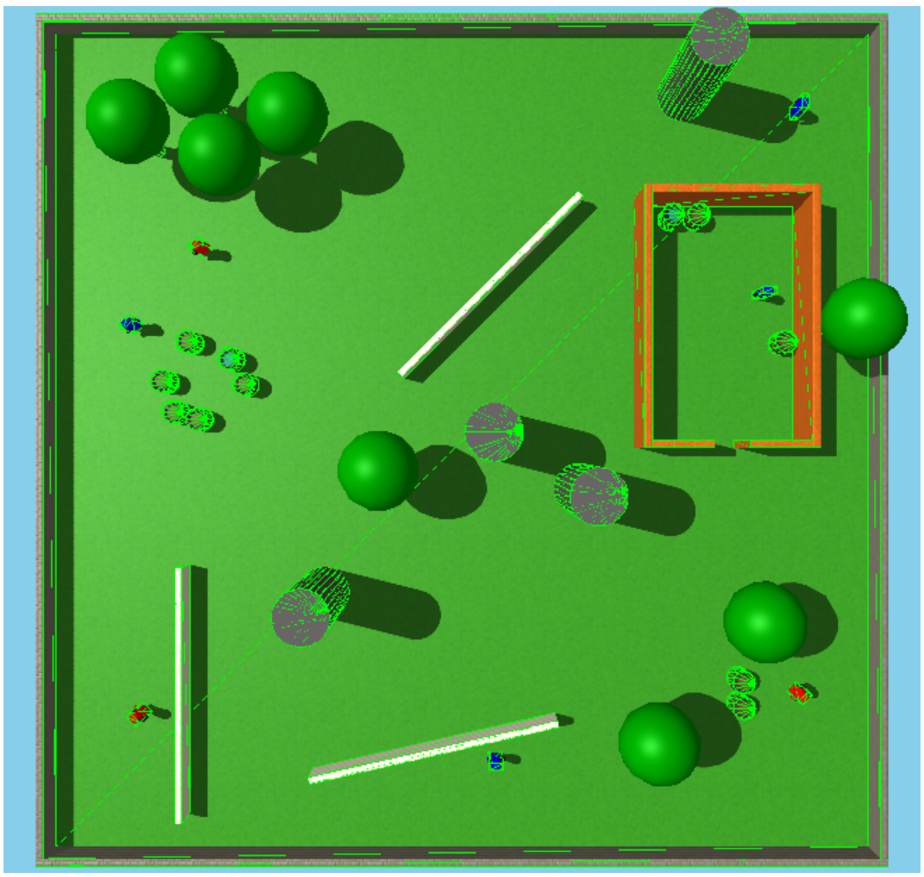

Figure 3: The map with the Cannon debugger seen from above