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. 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 `maps`.
- 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

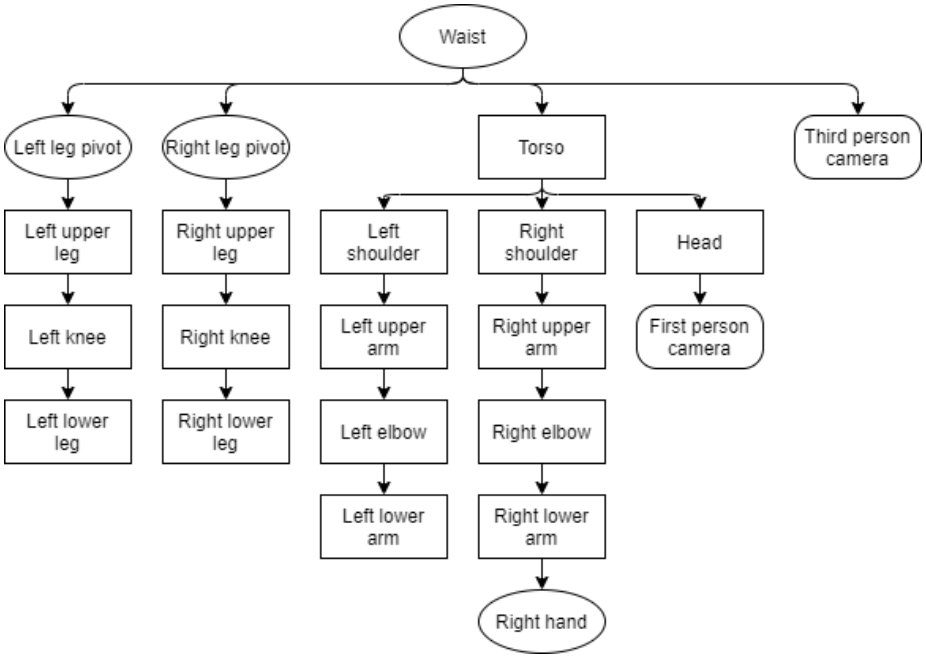

Figure 1: Robot's hierarchical model

4.4 Animations

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