# Low poly robots attack!

Interactive Graphics, AY 2019/20

Presented By:
**Andrea Napoletani**
**Sergio Picca**

Submitted to:
**Prof. Marco Schaerf**


SAPIENZA
UNIVERSITÀ DI ROMA

MSc in Engineering in Computer Science
June 2020

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Idea

*Low poly robots attack!* is the name of the final project of the Interactive Graphics course and is the first release of a 3D cartoon-style-game set in a sci-fi/futuristic world surrounded by big buildings in the desert. It is a first person shooter game where the user impersonates a character that must eliminate all the robots that want to kill him. The application is a survival game, so there are three different levels of difficulty *easy*, *medium* and *hard* that affects the life of the robots and their velocity. The goal is to survive as much as possible to the robot waves, each one is composed of seven little robots and one robot boss, which has the additional capability to shoot against the main character of the game and achieves the highest score.

We took inspiration from the old style games of the 80s, those from the Commodore and from the arcades, where young people could play. Moreover, particular attention was paid to the design of the game, trying to bring back the player in the past.

## 1.2 Project workflow

The workflow on the entire project is available on the GitHub. The development of the project is based on massive use of GitHub's facilities that allows creating a perfect organization for the developers. In particular, we used the `Project` tab on the repository in order to create a project, where insert the issues automated using the commits. In this way for any watcher or in general for those who are interested in our work is possible to see the various phases of the project and how the issues were solved, it also gives an easy way to keep track of the problems, bugs and things to implement.

For this first version, the project contains more than two hundred commits and thirty-two solved issues, by the authors Andrea Napoletani and Sergio Picca.

## 1.3 Technologies

The project is entirely based on *three.js* [1], a cross-browser JavaScript library and application programming interface (API) used to create and display animated 3D computer graphics in a web browser. Three.js allows the creation of graphical processing unit (GPU)-accelerated 3D animations using the JavaScript language as part of a website without relying on proprietary browser plugins. This is possible due to the advent of

WebGL (on which it is based). In the application we also used another library called Tween.js [11] for achieving smooth animations and for particular cases, in which it was extremely useful.

# Chapter 2

# Game environment and models

In this chapter we are going to explain the scenario in which the game takes place, in particular which are the various pieces used to build up the scene, including the imported models and the hand-made models, obtained by combining different geometries.

## 2.1 Landscape

An important part of our game is represented by the scenario where the game lives and all the actions take place. In order to create that, it has been used a technique called **Skybox** [2] and different imported 3D models. What we really care was avoiding to put some walls around just to delimit the area, creating a simple square where the hero can move, instead we really want to create something that resembles a real world, something with areas and spaces to explore.

### 2.1.1 Skybox

The skybox is a box that surrounds the entire world of the game. It is used to create a background that is not statics but can represent a 360° scene around the camera. It is done applying different textures to the 6 box faces and place the camera inside the box. All the game will take place inside the cube.

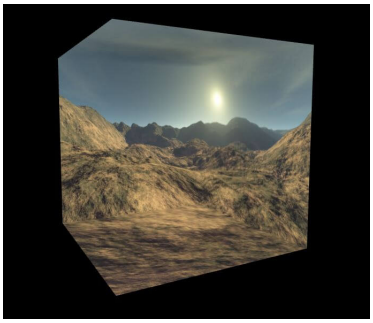

Figure 2.1: Skybox effect.

### 2.1.2 Floor

A floor has been created to contain the battle arena using a simple plane inside the box of the skybox. To obtain a realistic visual impact of the floor, a sand texture has been applied on all the plane. It is repeated many time both vertically and horizontally to

obtain a better result.

Each model of the game will move on this floor (bounded by the value of the $y$ axis) to simulate the gravity force undergone by the various models that are in the scene.

### 2.1.3 Models and map limits

The game arena is composed of more 3D models (each one composed of an *.obj* file and a *.mtl* file for the materials and textures application). They are loaded through the `OBJLoader` [3] of three.js. The load of these models is handled by the `LoadingManager` [4] tool of three.js that allows to load all the models at the start of the game and return to the user the percentage of the loading through a progress bar. In this way, once the game is loaded, all the models of the landscape are ready and the user can immediately start playing the game.

**Cities models** The scenario is composed of two main models: *Scifi Floating City* [5] and *Center city SciFi* [6]. These two models are positioned in a way that is used to bound the walkable area of the game.

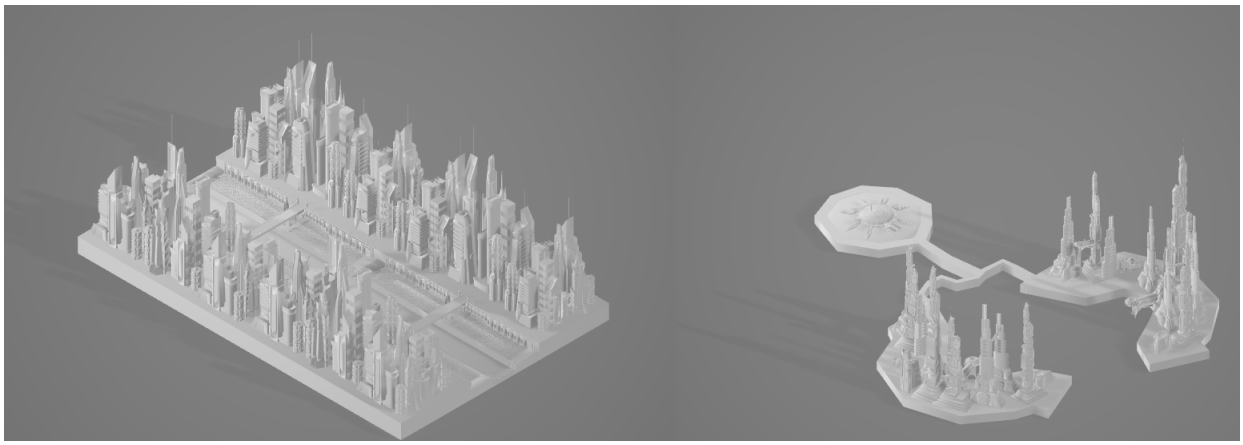

Figure 2.2: Cities models.

**Barriers** A model of barrier [7] is used to set the map limits that the character can't overtake. This model is loaded more time with a custom location in order to create an entire line of barriers.

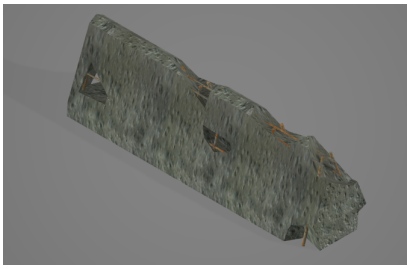

Figure 2.3: Barrier model.

**Map limits** The map limits are implemented exploiting the `Raycaster` [8] class of three.js, indeed it is used to create rays that recognize collisions of the main character among objects detecting the intersection of the rays starting from the centre of hero's bounding box to the vertices.

This method is applied in different aspects of our project, one of this is to limit the area in which the character can walk. A better description of this method is given in the Collision detection section. The logic is the following, when the character walks towards a wall or barrier, the rays of the boxes intersect and trigger a collision that will stop the movement of the character in the direction of the collision. Unfortunately, this solution has some drawbacks due to the low precision of the collision detection in some particular cases. Nevertheless, we can obtain great quality in limiting the area of interest.

## 2.2 Lights

In the scene are present two kinds of light sources: one *hemispherical* and one *directional*. The *hemisphere* light source allows illuminating the scene with color fading from the sky color (light blue) to the ground color (grey).
The *directional* light creates parallel rays of lights from a point in the world; it is used to simulate the light generated by the sun and its position is fixed according to the position of the sun represented by the texture images on the skybox.

### 2.2.1 Shadows

A system of shadows has been developed for all the models present in the scene. The shadow effect strictly depends on the chosen materials, in particular the `MeshToonMaterial` used resembles the *cartoon shading model* used in the first homework. This choice has been taken according to the retro style of our game. Furthermore, we enabled the `shadowMap` for the `renderer`, then we set to true the `castShadow` property of the directional light and for each object in the game we set to true the `castShadow` and `receiveShadow` properties.

## 2.3 Sounds

Regarding the sounds, we decided to provide many different effects to the game. In particular, we decided to apply sounds effects whenever the main character shoots by using its blaster and when he is walking, while a background music is provided both in the menu and in the page where the real game starts. A class `SoundManager` has been created and it loads all the sound by using the `loadSounds` function, where the `AudioLoader` of three.js is called, then once that every sound effect has been initialized the `play/stop` methods are called in the main game page depending on user actions. The sounds effects were taken by Freesound [9] website, they were adjusted for our application, for instance the blaster sound was cropped in order to be adapted to bullets' speed.

As soundtrack for the game, two different songs were selected, for the menu "Mammagamma" played by Alan Parson Project and for the game "Love is Anywhere" (FM Attack Remix) by Flashworx, the choice was due to the they resemble the 80s game-style of the project. The way the two songs are applied to the application is different, indeed the former was included in an `audio` element, while the latter was reproduced using the `Audio` class by three.js (like the sound effects).

## 2.4 Game models

The hierarchical models built by us used in this game were three: the main character `Hero`, the `KillingRobot` and the `RobotBoss`. Before describing each model into details, each of them has its own Javascript file, with functions creating the various parts (`createHead`, `createWheel`, ...). Every file has its own *sizes* object, where the measures of the various components of the model are specified.

- `Hero`. This object is the main character of the game, he has to shoot to the robots and survive as long as possible. The model is composed by thirty-nine meshes, the complexity is given by the fact that the hero has a weapon, having many different components and details. Even if the style of the application is very *cartoon oriented* and it is a first-person-shooter, we can distinguish different parts on the character, such as the fingers, target window on the gun and the final target element, used to shoot in a more precise way. Here the image showing the various parts of `Hero` and the gun model, that was too big to fit in one image. The *details* present on the gun refer to some `BoxGeometries` applied on its body to achieve some shining and glowing effect in order to give a sci-fi feel.

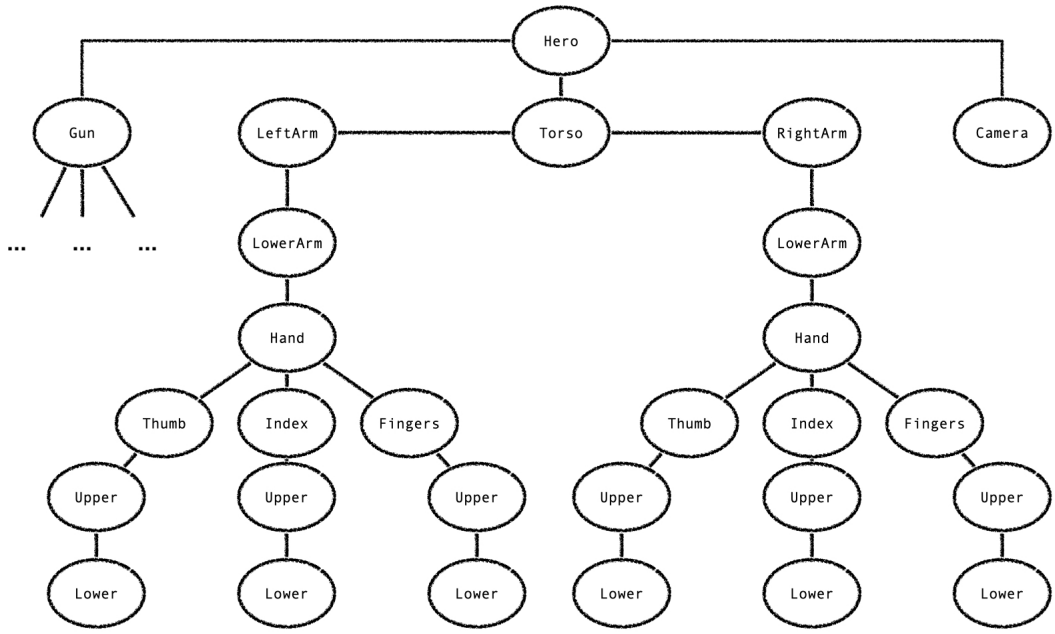


Figure 2.4: The hero model.

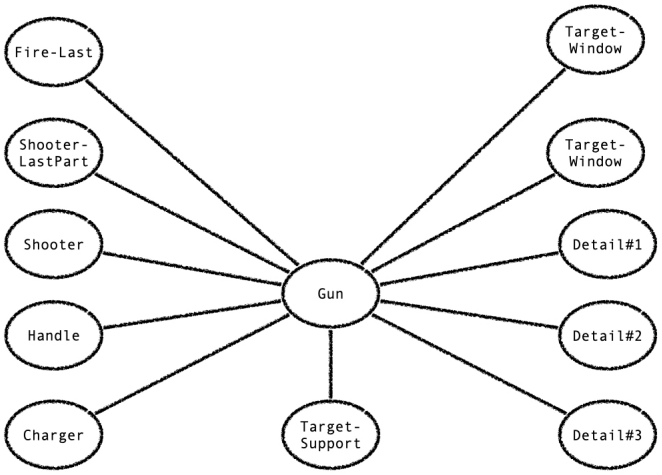


Figure 2.5: The hero gun model.

- `KillingRobot`. The killing robots are composed by twenty-two parts, there are many of them in the scene and they can hurt the main character, they have a cannon that is made up of three different parts, but they do not shoot any bullet due to an efficiency problem. The various parts are created as in the case of the `Hero` character by using dedicated functions, then they are attached to the object and all of them generate the various meshes according to the sizes stored in the object `robotSizes`.

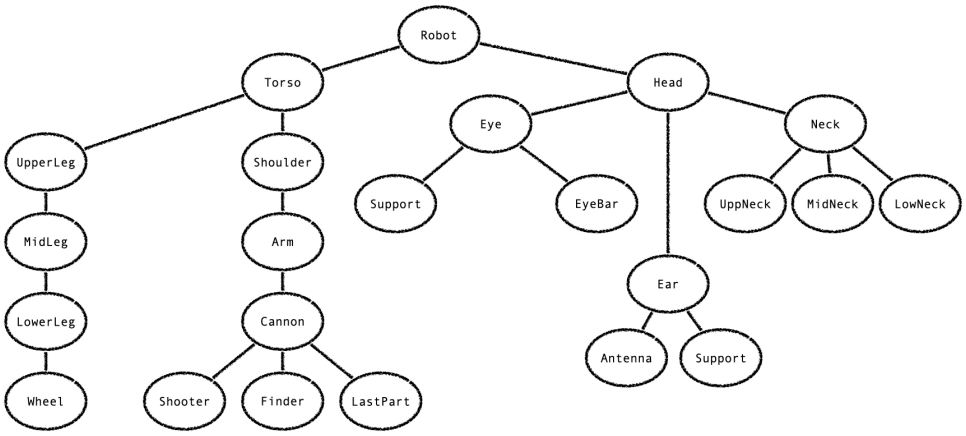


Figure 2.6: The robot model.

- `RobotBoss`. The robot boss is obtained by the `KillingRobot`, we just make it bigger, indeed it is five times bigger than a standard robot, and allow it to shoot to the main character. There is a little change in the colors, since the boss is bigger and has two ears and red antennas. A little change was on the head, in fact the radial segment parameter of the `CylinderMesh` was set to five. Furthermore, the boss has also some additional details on the torso, they are just box geometries with a shining color, allowing to distinguish the boss from the other robots.

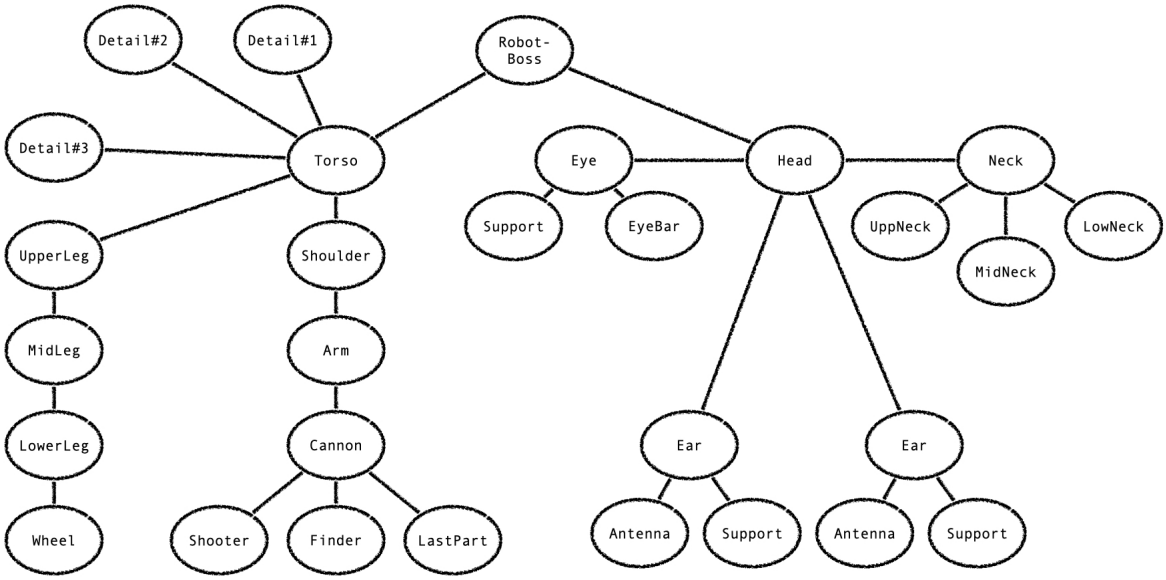


Figure 2.7: The robot boss model.

# Chapter 3

# Implementation choices

## 3.1 Animations

In this section we are going to explain how the various animations were created, in particular they were all **hand-made**. In order to achieve a better effect, we decided to use in some particular case the `Tween.js` library [11], in particular for the robots movement. In both robot and hero cases, a dedicated Javascript file was created, providing the functions triggering the animations, in order to get a modular and scalable solution.

### 3.1.1 A common color change

This could not be strictly considered a real animation, but influenced the look and feel of every character in the game. We were looking for a way to communicate to the user how he or she is damaged by the robots or damages them, **the solution was a color change depending on the life variable**. Furthermore, the color material of each robot's head and body changes, depending on if their conditions are critical or not, at the beginning they are grey or dark grey in the boss case, then once the life goes under one particular threshold the color goes to yellow, then if more damages are applied the color is orange, until they are going to die, in that case they became red.

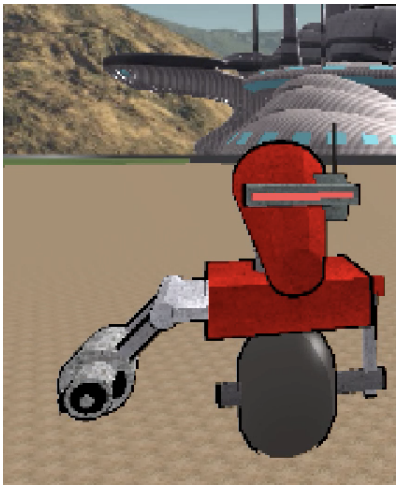

Figure 3.1: Robot is going to die.

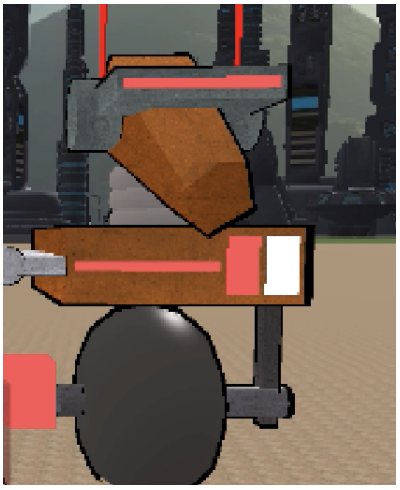

Figure 3.2: The boss is damaged.

The little robots and the boss have different life values, so the changes will be triggered according to different threshold values. The same technique is applied also to the hero, in this case only the upper and lower arms change color.


Figure 3.3: The hero has got a quite critical shoot.

### 3.1.2 Hero animation

The hero animation is provided by `AnimateHero` class in the `main-char.js` file, offering various functions handling the different movements of the main character, in particular there are there are five types of various animation.

1. **Walking animation**. This particular animation is triggered when the main character is walking, moving up and down the hero, it is obtained by increment and reduce the y-coordinate value, using the *sin* function, passing as input values multiple of $\frac{\pi}{16}$. Furthermore, in order of achieve a more smooth animation the function is multiplied by an attenuation function equal to 0.0025.

2. **Reload animation**. This kind of animation was particularly difficult to achieve because it requires to move the arm of the hero, with his hand and the gun handle, composed of many meshes. In order to correctly simulate a *reload movement*, once there are no more bullets, the starting position of the arm was necessary, then it was stored in a variable. The function is regulated by a boolean variable called `reloadFlage`, if true, the method decrements the arm (together with its meshes) and the gun reloads piece along the z-axis, until the current position of the arm was equal to the starting one decremented by one, then the flag is set to false, meaning that the reload was completed and everything is augmented to put all pieces as they were.

3. **Shooting animation**. The shooting animation is triggered once the main character is shooting, meaning when the left-click on the mouse is pressed. The animation is pretty simple, is triggered once the homonym method is called and it increments and decrements the position of the hero, still by using the *sin* function, but using numbers multiple of $\frac{\pi}{8}$. Then everything is multiplied by 0.05 to reduce the speed of the movement.

4. **Target position**. The target position is used to shoot in a better way and to aim robots, indeed this animation provide a different view, the difference can be seen in the following game snapshots.

   Considering that the main character has a camera attached to his root element, the animation is obtained by positioning the camera on the gun body once the SHIFT key is pressed. In the method called `targetMode` the future camera position is computed, because the character is moving, then it is adjusted a little and the camera current position is decremented along the z and x axes, while it is incremented along the y-axis, because having a higher camera allow the user to see better the bullets.

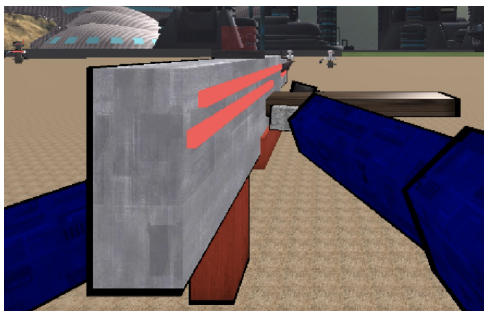

Figure 3.4: No target.

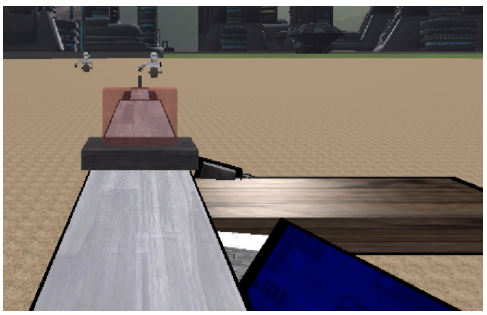

Figure 3.5: Target.

Once the SHIFT key is released, then the `returnFromTargetMode` function is called and it set the camera back to its position, the original value was previously stored in the class, under the variable called `cameraOriginalPos`.

### 3.1.3 Robots animations

The function used for the robots animation is called `AnimateRobot` and it is exploited also for animating the robot boss. In this case we decided to provide all the animations using Tween.js [11], because it is a more suitable and clean solution, especially for the robots movement. The parts that the robot will move are the head, the cannon outer part and the wheel. The effect of movement is achieved by incrementing and decrementing the rotation of the various parts, in particular they are incremented until they reach a particular value and after that they are reduced, until they are equal to the opposite value and then incremented again in a loop fashion. This final result is achieved by **combining two tweens**, using the `chain` method provided by Tween.js documentation [11], in particular once the first tween ends the second is triggered and once it finishes, the first is called again, this will repeat forever.

Instead of passing to the tweens directly the various parts' rotation, an object called `position` was created, it contains a value for the *cannon rotation*, the *head rotation* and the *wheel rotation*, then these values are assigned to particular coordinates of rotation vector of the various parts meant to move by using the additional methods present in the `AnimateRobot` function. These functions are indeed very simple, because they just assign the current rotation value of one mesh to the one incremented by the tween, for instance in the case of the robot cannon, `root.getObjectByName("robotShooter").rotation.y` is assigned to `position.cannonRot`.

Furthermore, every robot should also move toward the main hero, because they should attack him, this is also achieved by using a tween passing as input value to the method `to` the main character current position and because of that this tween is created in the render loop, inside the `animate` function. This is done for every robot, including the robot boss, but it has a different speed, in particular it is slower of all its *little children*, no matter what difficulty the user has selected, since the ratio among the two different velocities values is the same. In order to avoid that the little robots will overlap, because they have the same target position, we decided to mitigate the final value they have to reach by using a scalar, in this way they distribute uniformly like an army. The `lookAt` function of the robots was also modified, because they must look toward the main character, so we pass as input value the hero position along the x-axis and z-axis, since there is no

variation along the y-axis (the character cannot jump).

### 3.1.4 Bullets

The bullets exit from the robot boss cannon and the main hero weapon, they are just green and red *boxes* respectively. The idea came from a tutorial from the `saucecode`'s repository, called `threejs-demos` [12]. However many of the logic behind should be changed, in particular the one related to the bullet's *velocity*, because in the author's case there was no pointer lock. The logic is the following, given the XZ plane, since the character does not jump, we can consider the goniometric circumference draw by the hero rotation, then the radius is given by computing the *sin* and *cos* function respectively of the hero rotation along the y-axis. The following draw will explain it visually in a more clever way.

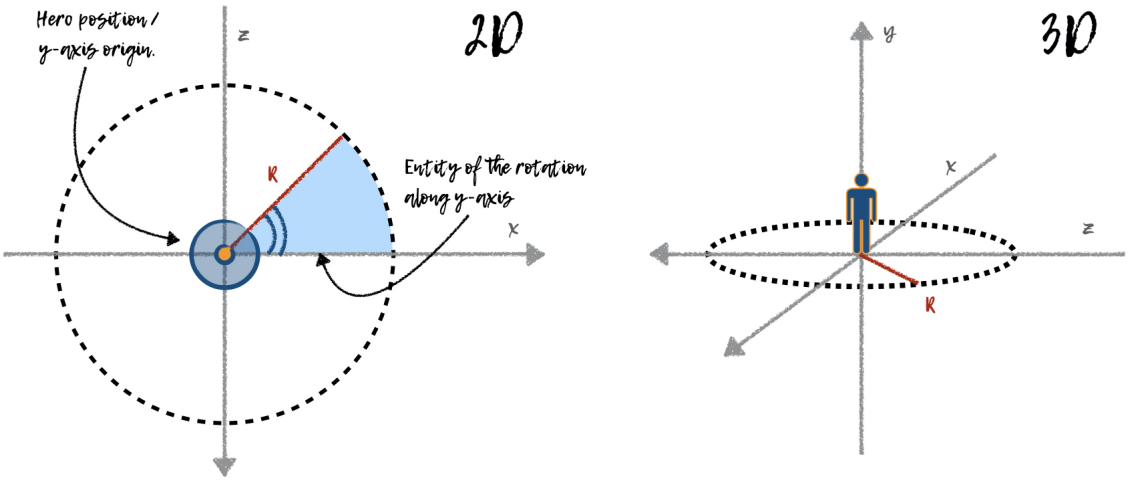


Figure 3.6: The bullet geometry.

Furthermore, calling the rotation angle along with y-axis $\alpha$ the equation of the radius $R$ is equal to $R = -\sin\alpha + \cos\alpha$, we need to put the minus in front of the *sin*, because we are pointing on the negative values of the z-axis. It seems that the direction of the velocity is now given, but the *issues of pointer lock* was that the rotation angle along y ranges on $[-\frac{\pi}{2}, \frac{\pi}{2}]$, so there was a point where the direction was correct, but the vector was pointing on the opposite side to the camera, so the bullets were shot in the wrong direction, because the angle went from 0 to $\frac{\pi}{2}$, then again back to 0 and finally negative, until it reaches $-\frac{\pi}{2}$.

**The issue was solved by using the** `getDirection` function of the `controls` variable returned by the `PointerLockControls` constructor, in this way the bullet was shot in the right direction.

## 3.2 Collision detection

Since in this application no physic engine was applied, because it would be an unnecessarily enhancement of the application as described later (see Drawbacks and bugs on Chapter 5), a very simple collision detection system was implemented, in particular two kinds of differnt collision techniques were applied.

### 3.2.1 Bounding box

The bounding box technique was applied by taking inspiration from the code by `stemkoski` [13], the working example is also hosted in his GitHub Pages website [14]. The collision detection is obtained by first adding an invisible `BoxGeomtry` to the `Hero` root element and second by using the `Raycaster` [8] constructor of three.js. The rays are computed for every vertex of the invisible cube, starting from the cube position to the vertex one, the near and far parameters are the default ones, in this way the rays extend to the *infinity*. We increased the number of vertices, with respect to the `stemkoski` example, in order to achieve a more precise collision detection, but it was not so good as the one provided by Physi.js for instance. The intersection with the rays is computed for all the object inserted in the `collidableList` array.

The other issue to solve is to understand which was the direction of the character while he was colliding, indeed if he was going straight, he should be able to go backward. However, in the first version of the collision detection system, this was not possible, so an object called `directionOfMovement` was created, with four components, one for each direction, that is incremented by one whenever the character is moving in the corresponding direction (if is going left, the left direction is incremented as long as the key for the left movement is held). Furthermore, four boolean variables were created, initially set to false and in the collision loop the maximum among the four directions is computed, so if the max value is equal to the right direction, meaning that **the last movement happened in the right direction, the right boolean variable is set to true** and the movement of the character is stopped on the right side. In other words, the `directionOfMovement` variable is just a way to remember which was the last movement of the character before the collision.

### 3.2.2 If too close, it hurts!

In this case the collision is computed just by checking if the distance between one object and the other was under a pre-defined threshold, by using the `distanceTo` method of the `position` parameter of every mesh. This kind of solution is **simpler** and **light weight** with respect to the collision loop of the previous section, indeed this technique is used for the bullets, both on the hero side and on the robot boss side for decrementing their health values, and for the position of hero and its enemies, indeed given the fact that the little robots cannot shoot, they **hurt our hero if they are too close to him**.

This solution also avoid to add extra invisible boxes for every robot and to do a collision loop for each of them, this would probably kill the performance, because the cycle should be implemented in the render loop and by doing so the final application cannot be played on many computers without a dedicated GPU.

## 3.3 Pixel post-processing effect

In order to give to the application a more 80s look, we decided to apply a post-processing filter by using the three.js's `EffectComposer` [15], in particular we selected the *postprocessing pixelate filter*. The final effect is achieved by tuning the parameter controlling the pixels size, indeed if the pixels were too big they will affect the game experience and do not allow the player to what is shooting at.

# Chapter 4

# How to play

## 4.1 User interactions

The game is based on interactions with the user through the mouse and the keyboard of its computer.

### 4.1.1 Menu

This first page allow the user to select the difficulty of the game. This change will affect the quantity of life of each robot and also their movement velocity.

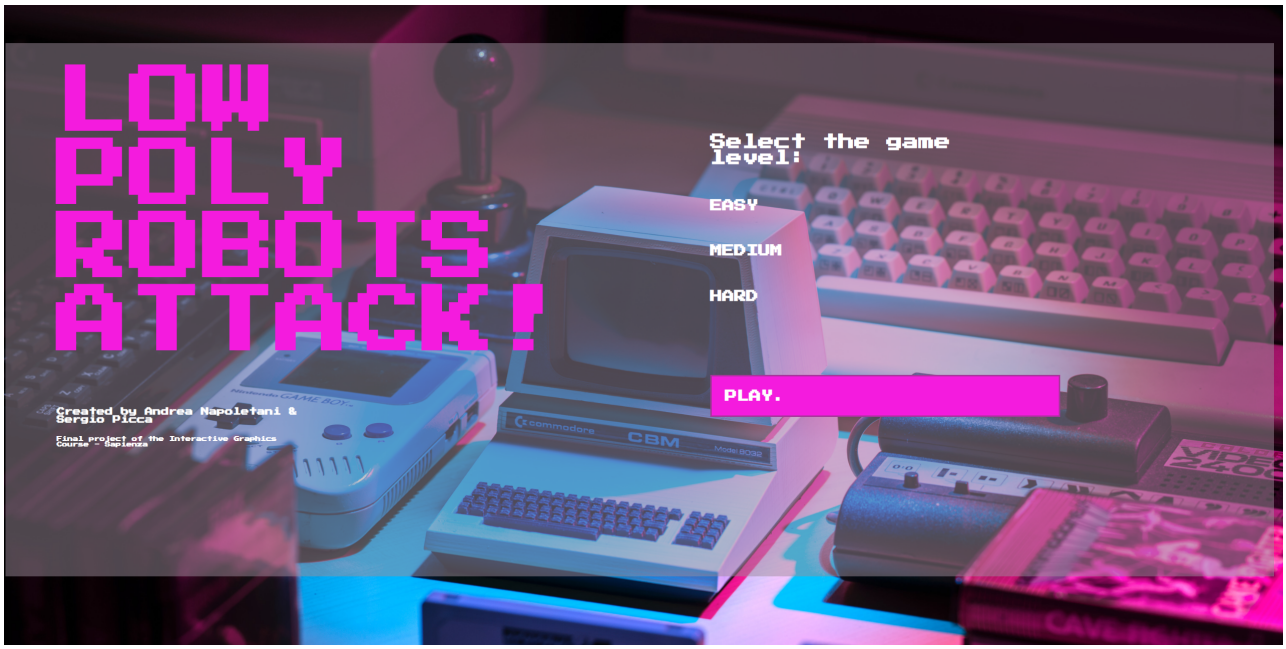


Figure 4.1: Menu.

### 4.1.2 Mouse and camera movements

Once the difficulty is selected the total control of the user's mouse pointer pass to the application that handle it to move both the character and the camera (that are parts of the same object) around the character. It is implemented exploiting the `PointerLockControls` [10] class of three.js, developed on `Pointer Lock API` (formerly called *Mouse Lock API*) provides input methods based on the movement of the mouse over time (i.e., deltas), not just

the absolute position of the mouse cursor in the viewport.
It allows the user to aim a robot simply moving the mouse in its direction and shot against him with the **left-click** of the mouse, like the most famous FPS games.

### 4.1.3 Keyboard and character movements

The keyboard allows the user to do some movements of the character in the game, in particular:

- move the character in the forward direction pressing **W** or the **UpArrow**;
- move the character in the backward direction pressing **S** or the **DownArrow**;
- move the character in the left direction pressing **A** or the **leftArrow**;
- move the character in the right direction pressing **D** or the **rightArrow**.

The double choice set of keys for the movement has been implemented to accommodate the user's needs.

All these movements have been implemented though translation of the character object on the `xz` plane.

There are also other two kind of interactions with the keyboard that can be triggered with the pression of one or two keys:

- *run* mode pressing **SPACE** + **W** or **A** or **S** or **D** allows to move faster in the selected direction;
- *aim* mode pressing **SHIFT** to move the camera over the gun and allow the user to aim with a different point of view.

## 4.2 Game logic

In this chapter will be explained the logic behind different tasks of the game, how the game proceed and continues. In particular, the general rule is simple, the player has **to survive as long as possible** to reach the highest score, as it was in the oldest arcade games.

### 4.2.1 Robots spawn

The robots and the boss are created at the beginning of the game, so the player will face his or her enemies from the beginning. The robots' positions are randomly selected in a range of values so they can appear also near the hero and hurt him from the very beginning, so the player should be careful to move quickly by using the letters, the arrows and maybe by using the bar, moving faster. There are at most eight robots per round, seven little and one boss and, as previously written, their speed and life change according the game difficulty.

### 4.2.2 Shooting

The main hero is able to shoot in the direction he is looking at, but be careful, because the munitions are not infinite, they are just twenty-four, and they are showed on the left side menu, so once the charger is empty, the player **just need to press R to reload**, but a message on the menu will notify him or her. The damage inflicted by the shoots is equal no matter what is the difficulty level selected at the beginning.

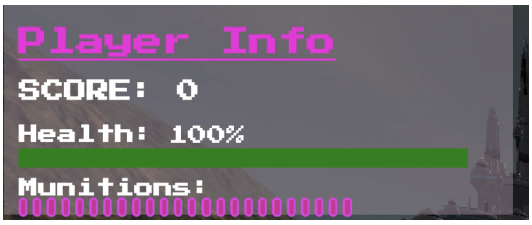


Figure 4.2: Munitions.


Figure 4.3: The notification of reload.

The robot boss will shoot in the direction of the main hero only when it is quite close to him, otherwise it will try to reach him to be able to shoot and the chase begins.

### 4.2.3 Life control and score

In order to win the game, the main character should survive as long as he can, meaning that his life should not be decremented. The life value of the hero is showed in the menu on the left and it is equal to one hundred. The lives of the little robots, depends on the difficulty value selected by the user, they range from six (easy case) to fifty (hard case, indeed they are pretty hard to kill!). The robot boss's life also depends on the difficulty level selected, it goes from forty to one-hundred as the difficulty is increased.

The second game's object is to achieve a very good score, the higher the better, then depending if the the hero hits or kills one of the robots the score value is incremented and it changes the color being green until the update ends, while if the main character is hit the score is decremented and its color switch to red as showed in the two figure below. The entity of the increment or the reduction depends on which robot is hit or hits the hero.

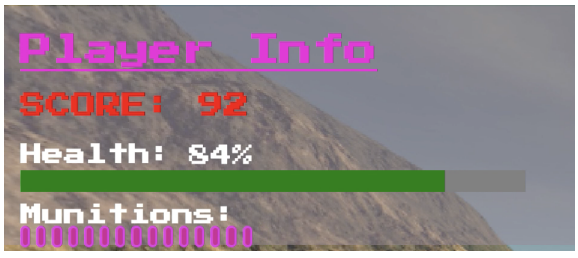


Figure 4.4: The hero is getting damaged.


Figure 4.5: The hero is hitting robots.

# Chapter 5

# Drawbacks and bugs

In this chapter will be justified some technical decision taken in the developing of the project and will be also discussed some bugs present in the actual version of the game.

First of all must be explained the decision to not use a physical engine to simulate the physics of the game.
Initially the models of the main character and the robot have been created using *three.js* (see section Game models). The "translation" of all these models from *three.js* to a physical engine paradigm like *Physijs* would has been a time-consuming operation especially in the case of very complex model like ours. Moreover, the use of `PointerLock` for the mouse interaction would have made the task even harder to apply. So, given the minimal use we would have made of it in comparison to the difficulties in applying it, we opted for a different solution.

## 5.1 We know our limits

Given the fact that it is the first release of the working project, we are aware of some bugs present in the game. Develop an FPS game is an heavy task from different points of view, including coding difficulties and hardware performance management. For this last reason some initial choices have been reviewed to adapt the workload to the execution on a browser and obtain a good gaming experience for the user. The result is a trade off among graphical quality and playability.

### 5.1.1 Major bugs

One of the major problems is represented by the map limits due to the low precision of the collision detection in some particular circumstances, in fact, sometimes the walls/bushes doesn't stop the character that can overtake it.
Another problem that might occur (quite rarely) is that the character doesn't die even if its life il 0% and can continue playing like an *highlander*.

# Chapter 6

# Future improvements

The possibility to increase the complexity of the project is infinite. Taking as an example some of the most famous FPS titles can be an inspiration to implement new features of the game. Obviously, the more the improvements difficulty is, the more is the time to spend on.
Let us describe some features that increase the quality of the game, dividing them in two categories.
Technical improvements:

- implement physics in game using a physical engine to improve the interaction quality among objects;
- improve the precision of the shooting animation;
- optimize resources in order to obtain a smoother game.

Game experience improvements:

- add more guns and abilities of the character (i.e. throw a grenade);
- add the possibility to choose the side in the game (robot or character);
- add multiplayer ranking management to compare the results with other online players;
- change among different battlegrounds.

# Chapter 7

# About project

As said in the *Project workflow* the project is available on GitHub where is also possible to see the steps of our work during over time. The game is also deployed on GitHub pages to be accessible online.

The GitHub repository and the live demo of the game are accessible through the two buttons below.

**The authors** We are two students of MSc Engineering in Computer Science at 'Sapienza' University of Rome. This project is developed for the course of Interactive Graphics. Usefull links:

- **Andrea Napoletani** - [GitHub] [LinkedIn] [email]
- **Sergio Picca** - [GitHub] [LinkedIn] [email]

Feel free to contact us and give us some advice about the project.

# Bibliography


[1] Three.js
Javascript 3D library - https://threejs.org/

[2] Skybox example
Three.js Fundamentals - https://threejsfundamentals.org/threejs/lessons/threejs-backgrounds.html

[3] OBJLoader
Three.js docs - https://threejs.org/docs/examples/en/loaders/OBJLoader

[4] Loading Manager
Three.js docs - https://threejs.org/docs/api/en/loaders/managers/LoadingManager

[5] Scifi Floating City Model
Free3D - https://free3d.com/3d-model/scifi-floating-city-58552.html

[6] Center City Sci-Fi Model
Clara.io - https://clara.io/view/505922dd-0c99-4b7d-8374-51d008a7d230

[7] Old Road Barrier
Free3D - https://free3d.com/3d-model/old-road-barrier-98608.html

[8] Raycaster
Three.js docs - https://threejs.org/docs/api/en/core/Raycaster

[9] Sound effects
Freesound - https://freesound.org

[10] PointerLockControls
Three.js docs - https://threejs.org/docs/examples/en/controls/PointerLockControls

[11] Tween.js - https://github.com/tweenjs/tween.js

[12] Saucecode threejs-demos repository - https://github.com/saucecode/threejs-demos

[13] Stemkoski github repository on three.js - https://github.com/stemkoski/stemkoski.github.com/tree

[14] Stemkoski three.js working examples - http://stemkoski.github.io/Three.js/

[15] EffectComposer - https://threejs.org/docs/examples/en/postprocessing/EffectComposer