Interactive Graphics Project

Alessio Parmeggiani - 1799327
Alessio Sfregola - 1798423

Academic Year 2020/21

Contents

Introduction

The following is a report regarding the final project for the Interactive Graphics class at Sapienza University of Rome in 2020/21.

1.1 Game

The project consists in a simple game made using Babylon.js, a 3D engine based on WebGL. The game is a reimagination of the all time classic Space Invaders. The player impersonates a mech robot which has to defend the planet from a number of alien enemies, without succumbing in the attempt. The player can walk around the planet and shoot rockets to eliminate their foes, which spawn at the beginning of each round. If he manages to clear a level, the player is rewarded with a randomly chosen upgrade, or the chance to increase his remaining health. The player can win the game if they manage to clear out five waves of enemies, or lose the game if they run out of health.

1.2 Tools used

We used a large variety of tools to create our project:

- Javascript, a web programming language we used to create the backbone of our game.
- HTML, the standard markup language used in web development, and CSS, a stylesheet language, to create and customize the responsive UI of the game.
- Blender, an open-source 3D software we used to create all the models present in the game and their materials.
- Adobe Illustrator, a vector graphics editor we used to create some of the elements of the user interface (such as the power up icons).
- Adobe Photoshop, a raster graphics editor we used to edit some of the images used in the game.

Models and materials

We used Blender to model all the objects that are used in the game.

2.1 Player

For the player we modelled using Blender a tank with four legs, and each one of them is composed of 3 joints. The first joint, the shoulder, moves only horizontally (along the local Z axis) while the other two only along the local Y axis. The model also includes a cannon and the body is divided into two parts for a total of 15 parts. We followed a hierarchical approach and the parent object is the upper body, the other parts follow the graph below:

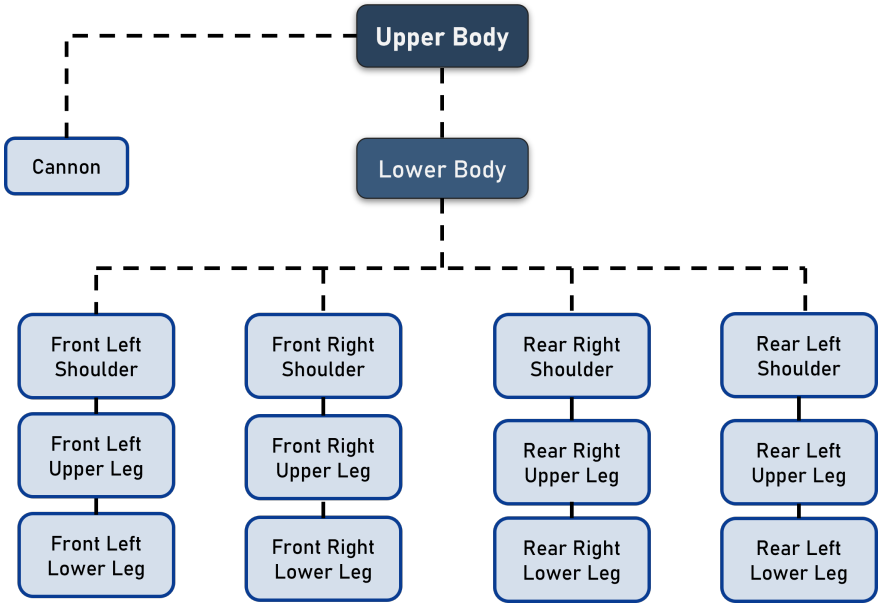

Figure 2.1: Hierarchical graph of player model

The model also has 5 different materials applied with 4 main colours and different types of metal on each material. On the render below I used some noises to give more variation to the colours but in the game we used a single colour for each material, otherwise at the start of the game we should load each texture and the game would be slower.

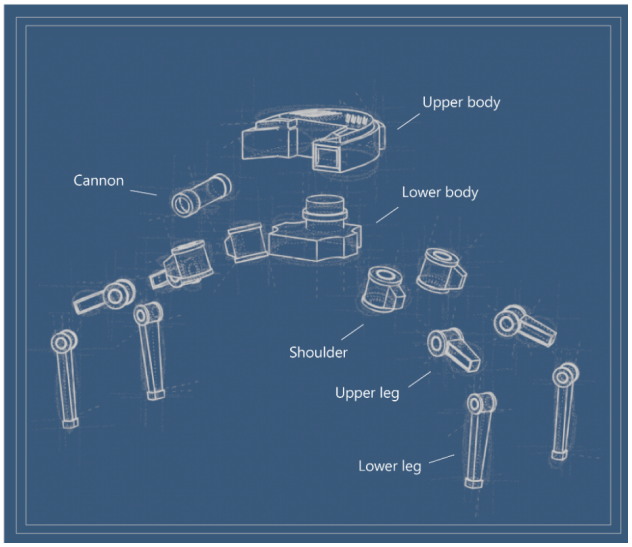

Figure 2.2: Blueprint-style render of the model, in which its parts are highlighted and annotated

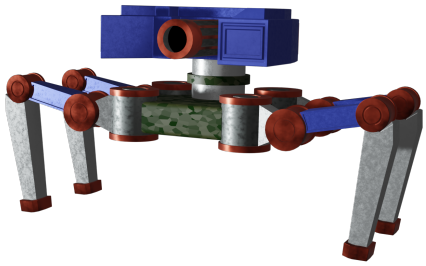

Figure 2.3: 3D render of the player model

2.2 Enemies

In Blender we also created 3 other models to represent the different types of enemies.

Figure 2.4: 3D renders of the three types of enemy models

The normal enemy has 6 different materials: a plastic material, two metal one with different colours, a partially transparent material for the dome and two materials for the different lights, blue and red.

The fast enemy has 4 materials: two types of plastic material, a green light and a glass material for the dome.

Also the tank type of enemy has multiple materials, in fact it has 5 of them; as in the other enemies there is a glass material that is almost transparent for the dome, a light, and 3 different metals with different values of metallicity.

All these materials have only a diffuse colour and no textures.

These models are loaded only at the start of the game, the original models is then disabled and thus invisible. When a model is needed it's not used the original model but is created an instance, in this way is possible to exploit hardware accelerated rendering and have many instances of a model without slowing down the game. The only limitation we found is that it's not possible to modify the material of an instance.

2.3 Other objects

For a visually appealing planet we created two more models: a plant model and a cloud model. Both have a single material and the cloud has an alpha value so is slightly transparent. The last model we created is the missile that the player shoots and is composed of two simple materials. Also for these models we used the same approach of the enemies: the original model is loaded and disabled, and instances are created on the fly.

Figure 2.5: 3D renders of the other objects: rockets,plants and clouds.

2.4 Planet

The planet is a simple sphere of variable size.

The material for the planet is a PBR material composed of three textures: a diffuse texture for the general colours, a glossiness texture that is used in order to have the parts with water more reflective and a bump texture. All these textures have been generated using Blender where the planet has been created with a procedural material. The diffuse colour was generated by using a noise that combines a tiled grass texture with a blue colour. Another noise creates variation of green on the grass part and a particular type of noise called Musgrave creates a wave-pattern on the sea part. By setting a threshold on the noise we also created a sand effect on the borders of the sea.

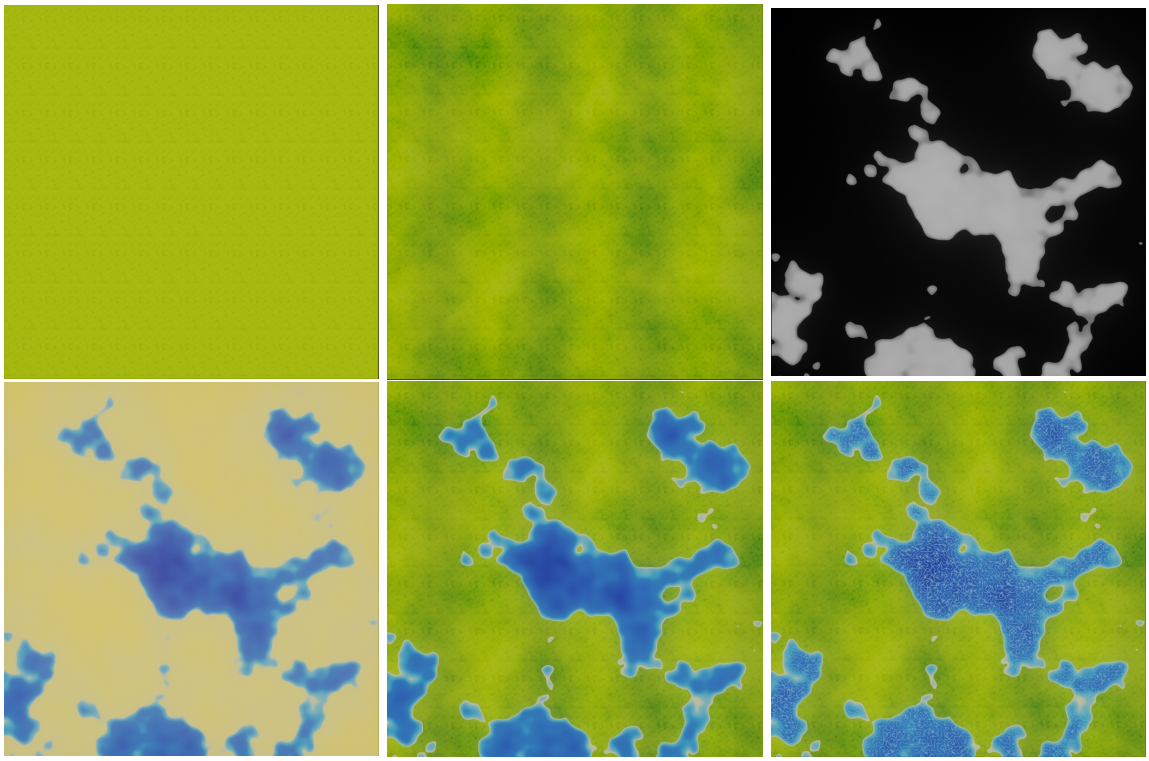

Figure 2.6: Some steps of the creation of the planet texture.

To extract the bump map from the grass texture I used some preprocessing using Blender and extracted a black and white version of the original grass texture, then used some noise to generate a little bump also on the water parts of the material.

Figure 2.7: The final texture for the color (left), roughness (center) and bump (right).

When the final look of the textures has been achieved we generated the texture for the entire planet as a normal png image with 2K resolution by baking the procedural material onto a sphere. In this way we extracted each texture by considering during the baking only a feature of the material each time.

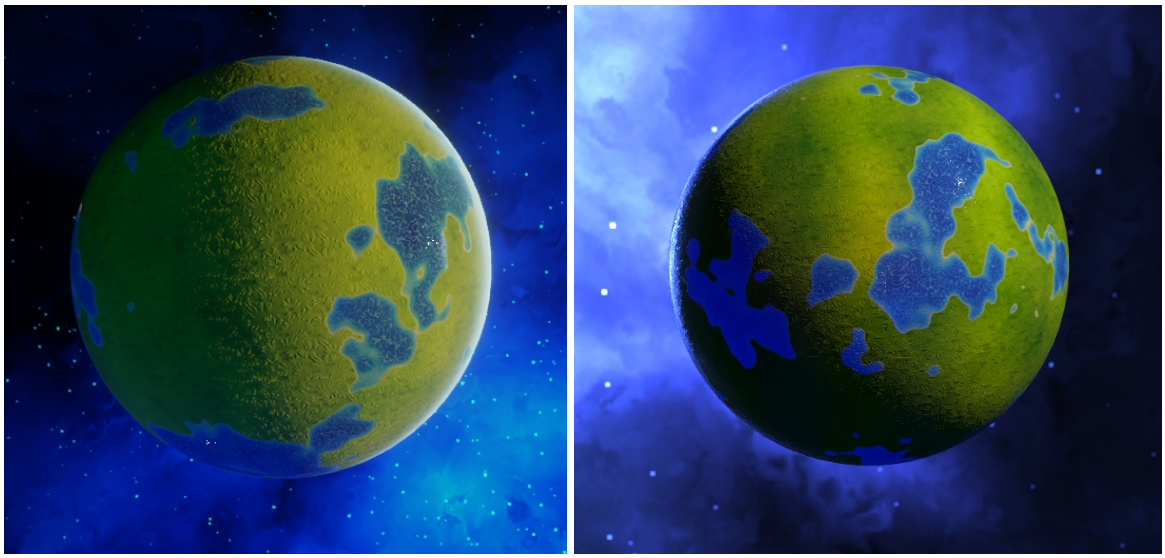

Figure 2.8: Comparison between the final material for the planet on Blender (left) and Babylon (right)

Environment

3.1 Populating the planet

To give the planet a more appealing look we also spawned in random points of the planet plants and clouds. To compute a random point on the surface of a sphere we used the formula:

$$(x, y, z) = \left(\frac{x \cdot \text{radius}}{\sqrt{x^2 + y^2 + z^2}}, \frac{y \cdot \text{radius}}{\sqrt{x^2 + y^2 + z^2}}, \frac{z \cdot \text{radius}}{\sqrt{x^2 + y^2 + z^2}} \right)$$

Where x,y,z are random points between -1 and 1.

Each plant or cloud has a different size, rotation and in the case of clouds, a different distance from the planet. This formula will be also used later to compute the spawn point of an enemy.

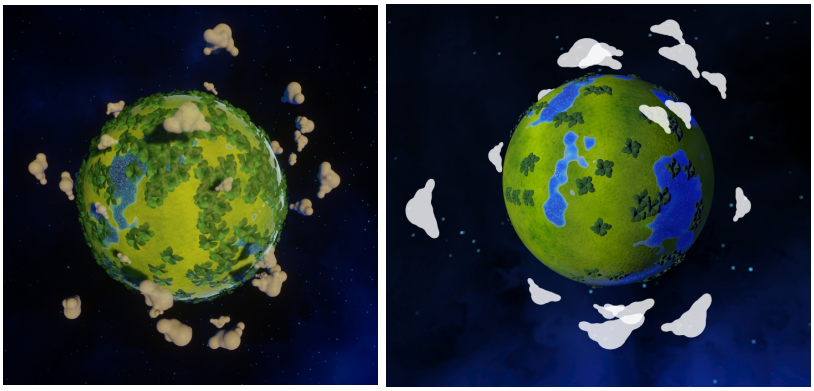

Figure 3.1: Comparison between the final planet on Blender (left) and Babylon (right)

3.2 Skybox

The skybox we used is a modified version of a skybox depicting a space background found on the web, the original skybox has a red tone on half of it and was too bright, so to maintains a more natural color we changed the red channel of the image using Adobe Photoshop and gave a blueish colour also to the red parts of the image.

3.3 Lights

To light the scene we used 2 different directional lights coming from opposite directions, one has a yellowish color and is the main source of light, the other has a blue color and is weaker, it's used mainly to give more vibrant colours to the scene and to softly light the dark face of the planet.

We decided to not include an hemispheric light in order to have a dark and lighted part of the planet and in this way there are also more distinct shadows.

The player has also headlights modelled using a spotlight, they are mainly useful when roaming the dark part of the planet.

3.4 Shadows

To generate the shadows on the planet we set up a shadowGenerator that reacts only to the main light and has a resolution of 2048. The objects that cast a shadow are all the parts of the player, the enemies, and the plants. The shadow generated by the clouds never fall on the planet so we didn't consider it. The only receiver of the shadow is the planet, in this way the shadows aren't drawn also on the other objects and the performances don't decrease.

Game mechanics

4.1 Enemies

In order to move the enemy toward the player we obtain the direction that link the enemy position and the player position by subtracting and normalizing the two positions; then a pivot is created and rotated to follow the computed direction, the pivot is then parented to the enemy mesh, and now by rotating the pivot along the direction the enemy will rotate following the surface of the planet and will go toward the player position. We included 3 types of enemies in the game: The most common one moves continuously and follows the player, it updates the position to reach if the player moves or after 0.5 seconds. To avoid clipping and to have a different trajectory for each enemy, the position to reach is not exactly the position of the player but it's a random point on a radius with the player position as center.

The second type of enemy is smaller and faster, it's behaviour it's not so predictable and it changes position rapidly without much accuracy, it also has less health.

The third type of enemy is the most rare, is the enemy with the more life points and is the slowest. It follows the same AI of the normal enemy.

The game starts with 3 normal enemies, after every wave is completed there will be 3 additional enemies and the chance of a different enemy will increase each time.

To represent how much life is left to each enemy we used an health bar object for each enemy that is a child of the enemy mesh, it consists of a cylinder with some glow and the size is reduced as the life decrease, in fact when the enemy is hit the health bar is scaled along the local Y axis and the new size is computed using:

$$\text{size} = \frac{\text{remainingLife}}{\text{fullLife}}$$

There are also two thresholds that determine the color of the health bar, when the enemy reaches half of the health the bar becomes yellow, and when the enemy has a low health the bar becomes red.

During the game, when an enemy hit the player, a part of the player health is removed and the player is now invulnerable for about 0.5 seconds, after this time it can be hit again.

4.2 Rockets

The rockets,like the enemies, move using a pivot: every time the player shoot a new rockets is created slightly forward with respect to the player and a new pivot is created in the center of the planet with the same rotation along the Z axis of the player, then every frame the pivot rotate of a value along the local X axis and so also the rocket rotates following the surface of the planet.

The rockets also fall to the ground after some time: in fact every frame each rocket is moved of a certain quantity toward the ground, when the distance between the planet surface and the rocket is lower than the height of the rocket mesh, both the rocket mesh and the pivot are deleted from the game.

The amount of distance it travels downwards each frame depends from the amount of revolutions around the planet that the rockets must perform before falling to the ground, it is computed using this formula:

$$x = \frac{-\text{height} \cdot \text{speed}}{2 \cdot \pi \cdot \text{N}}$$

Where N is the amount of revolutions.

By picking certain upgrades the player will be able to shoot multiple rockets each time, in particular it is possible to shoot multiple parallel bullets or more diagonal bullets.

If the bullets are parallel the position of the most left bullet is computed using:

$$\text{pos} = -\frac{(\text{bulletCount} - 1)}{2} \cdot \text{offset}$$

and each subsequent bullet has a position shifted of an offset value that is computed using the size of the rocket.

In the case of diagonal bullets the position remain the same but change the angle and rotation for the most left rocket is:

$$\text{rotation} = \text{originalRotation} - \frac{(\text{bulletCount} - 1)}{2} \cdot \text{angleOffset}$$

Where originalRotation depends on the rotation of the player and angleOffset is a fixed value. Each subsequent rocket will have the direction increased of angleOffset value.

To simulate the thrusters of the rockets we used a particle system with as emitter the rockets. We override the default start position for the particles in order to be able to create them from the base of the model and not from the center. There are 3 different types of particles generated with 3 colours: red,yellow and grey; and their size and lifetime is randomized.

The number of particles depends on the amount of projectiles, in this way the game is not slowed down if there are too many projectiles in the scene.

Figure 4.1: The particles on a rocket.

4.3 Upgrades

To make the game a little more interesting we chose to include the possibility of upgrading the player's abilities. The statistics that can be improved are:

- Player's speed: Increases the player's velocity by a factor of 1.5.
- Player's health: Gives the player an additional unit of health equal to 1/5 of the original health.
- Bullets speed: Increases the speed of the bullets by a factor of 1.5 and increases the ratio of bullets fired by a factor of 1/5.

- Bullets +1: Lets the player shoot two bullets at a time.
- Arc bullets: Adds two more bullets to the player's arsenal and shoots them diagonally.
- Bullet range: Increases the distance travelled by the bullet before falling on the ground by 50%.

Figure 4.2: Icons we made to symbolize the upgrades, in the same order as they are listed above

The upgrades are presented to the player three at a time. Two of them are randomly chosen, the third one is always the player's health. At each round the player must choose between increasing their chances of surviving by taking advantage of additional health, sacrificing their offensive power, or going all-in trying to defeat all the enemies before being overcome. Once an upgrade has been selected its icon is shown in the top-right corner of the screen and it can't be chosen again.

The actual implementation of the upgrade is straightforward, it consists of a callback function executed each time an upgrade button is pressed, that changes one of the global variables that store the player's and bullet's data.

4.4 Animations

To make the game feel more alive we added a great variety of animations to entities that populate its world. Let's start with the player model. The walking animation involves the 3 components of the leg. Since the legs don't move all in the same way, we used some if conditions to determine which leg we are referring to, and change some parameters accordingly. As an example, every shoulder performs a rotation of 40° around the resting position, but the actual values of this rotation change for every leg. At the same time:

- the front left leg goes from 25° to -15°
- the front right leg goes from 15° to -25°
- the rear left leg goes from -15° to 25°
- the rear right leg goes from -25° to 15°

The same happens also for the other two joints.

Other than this, the animations adapt according to the speed of the player, and its direction.

All of this has been obtained with a small number of parameters that change in just four if conditions exploiting the hierarchical structure of the model.

The legs are not the only moving parts of the robot. We added a very subtle downward animation of the body, to simulate the weight of the robot bringing it down when it makes a step forward. We also added an animation to the cannon, which is executed every time a rocket is launched. The cannon slightly retracts to simulate the recoil, and rotates to simulate the reloading of ammunition. To avoid a misplacement of the cannon when the animation is finished we hardcoded a reset position to reach every time the animation is finished.

The moving animations of the mech are triggered by the startAnimation() function, which starts the animation for all of the legs, calling addAnimation() four times, and bodyAnimation(). This is called when the user presses one of the WASD keys. When these keys are released, the stopAnimation() function is called, which brings all of the joints back to their original position.

To make everything smoother and more pleasing to the eye, we enabled the blending of all of the animation, so that when an animation stops it doesn't abruptly pass to another one, but smoothly transitions towards its new state.

When an enemy spawn it start a special spawn animation,a cylinder and two torus are created in the spawn position and a blue material is applied, it has an opacity texture that blend between an higher opacity at the base of the cylinder and a lower one to the top, in this way the enemy is still visible inside the cylinder. These objects start fully visible and gradually fade until they are no longer visible and they are deleted when the animation ends. During the fading the torus also rotates along different directions and with different speeds to simulate a teleportation effect.

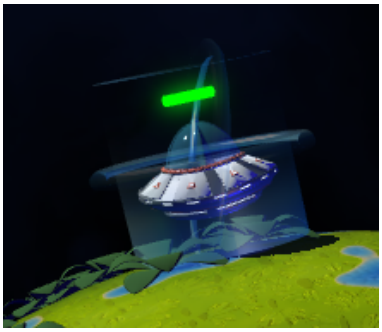

Figure 4.3: The teleport effect on an enemy.

There are also two simple animations for the rockets and the enemies, while moving both rotates continuously along the local Y axis. The last animation is used for the death of an enemy, when the health of an enemy reaches zero, the enemy begins to fall to the ground, so at each frame it's locally translated downward and slightly rotated. During the fall it cannot hurt the player and when it reaches the ground is destroyed.

4.5 Sounds

To increase the immersion of the user we added some sounds triggered by some of the game's events. These are:

- Rocket launching
- Player walking
- Enemy being hit
- Player being hit
- Completing a level
- Starting a new level
- Enemy dying

User Interface

The entirety of the user interface in the game is made in HTML and CSS. All the animations for the interfaces are made in CSS; they are triggered by removing and adding classes to HTML components. The interfaces are also responsive to the size of the screen, so the user can play the game on a variety of devices. There are four main areas in which the user directly interacts with the interface, which are the following:

Figure 5.1: The splash screen. It displays a rendering made in Blender of the planet in which the game takes place, the name of the game, a button to start the game and the commands to interact with it. The planet and the background are two different images, so that the planet's dimension can scale independently of the other components.

Figure 5.2: The game over screen, shown when the player runs out of health. It appears with a fade-in animation. The eclipse enters the screen slightly later than the rest of the components. From there, the user can start a new game.

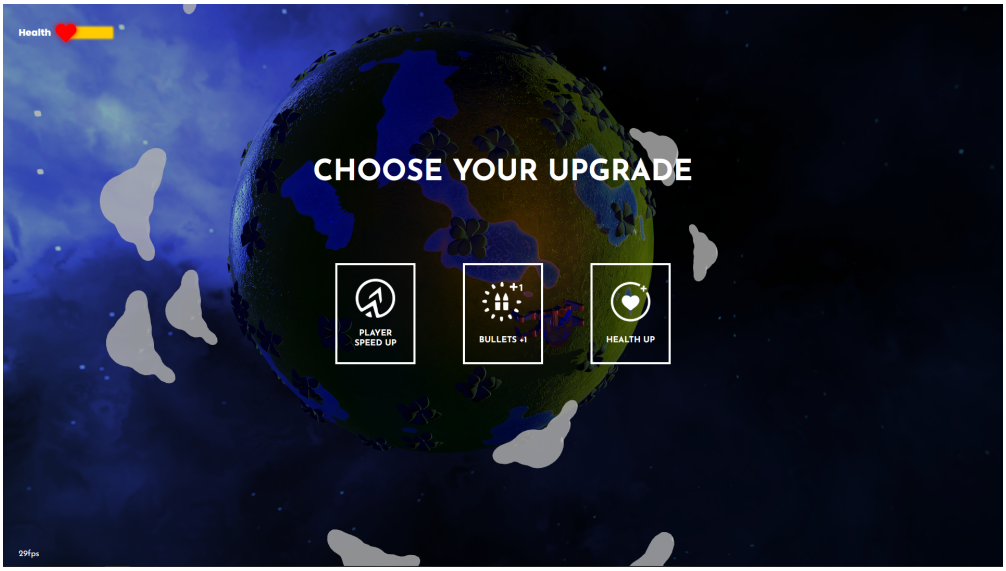

Figure 5.3: The upgrade screen. From here the user can choose between two randomly selected upgrades that boost the player's statistics or one unit of extra health. From here the health bar displaying the user's remaining life is visible. It changes color based on its percentage.

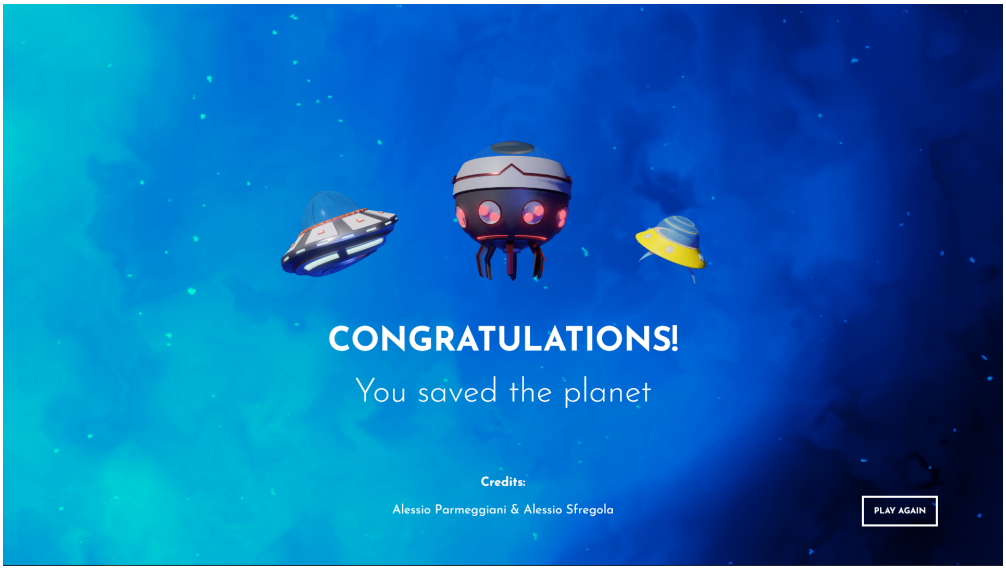

Figure 5.4: The end screen. The user sees this fading in when they finish the game. By pressing the button in the bottom-right corner the player can start a new match.

References

- Blender
- Babylon js
- Adobe Photoshop
- Adobe Illustrator
- Grass Texture
- Space Background