Interactive Graphics – Final Project

Funky Robot

Vincenzo Claudio Casanova, 1662174

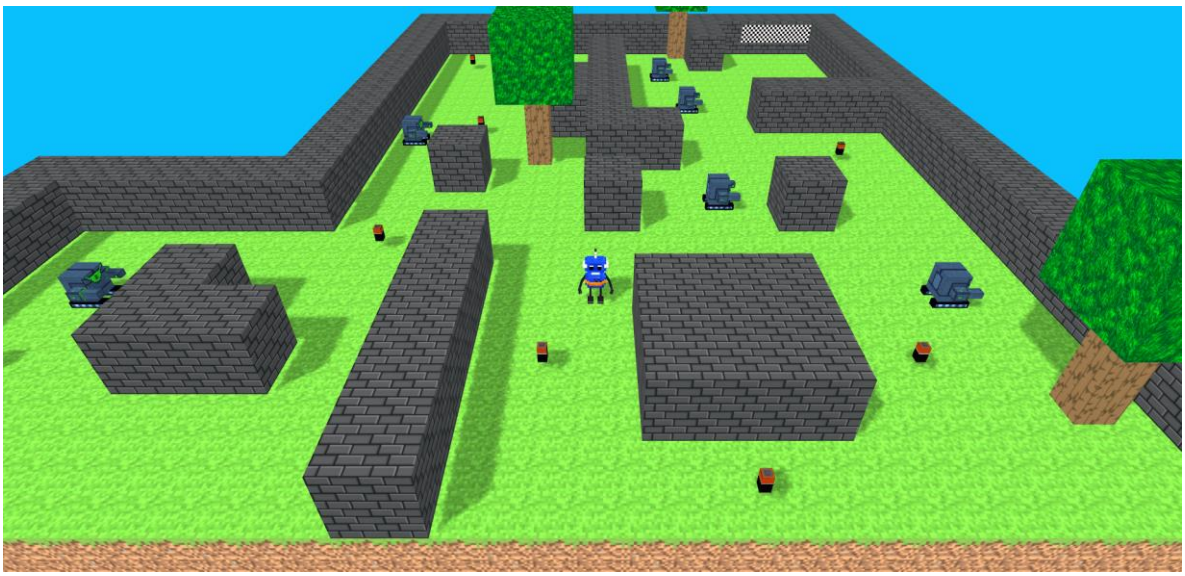
1. Introduction

Funky Robot is a game in which the player controls a little robot that needs to reach the end of the map in the fastest way possible. The game is fast to play and aims at the improving of the time score to make it fun.

Command and mechanics of the game are simple, the player can move in the four directions, and the map is populated by enemies that if touched will end the game, and little batteries that give to the player a speed boost for few seconds.

The game is implemented using Three.js as main library, Tween.js for animations and Physijs for general physics and collisions. Other libraries used are GLTFLoader and OrbitControls, that will be described later.

2. Scene



The scene is composed of a main platform (the ground) composed of two blocks, walls used to delimit the map and create different paths in the game, trees, batteries and enemies.

A transparent wall is used as the lower limit, to prevent the player to fall of the game platform.

The finish line is a plane geometry.

Camera and lights

The camera is perspective and centered in the player position, in a third person perspective, so that the player is visible towards the entire map.
The *lookAt()* function is used in the *animate()* function to update the camera to follow the player.
At the end of the game, when the player reaches the final line, the camera become orbital and the user can look around dragging the mouse and zooming in and out. The chance to look around lasts for few seconds, until the finish page is displayed. To implement the orbital camera, the OrbitControls library has been used.
In the *animate()* function (main.js), a control using the *orbitControls* flag is performed to recognize that the user has arrived at the finish line and activate and update the orbit camera, instead of the normal camera.

```
651         if(orbitControls) {
652                 controls.target.set(robot_box.position.x, robot_box.position.y, robot_box.position.z);
653                 controls.update();
654         } else{
655                 camera.position.set(robot_box.position.x, 11, robot_box.position.z+15);
656                 camera.lookAt(robot_box.position.x, robot_box.position.y, robot_box.position.z);
657
658         }
659         renderer.render(scene, camera);
660         requestAnimationFrame(animate);
```

Two different lights have been used, ambient and directional, shadow has been enabled.

The background of the scene is simply a light blue color that reminds the sky. 3D clouds would have been a nice add, but a lighter scene has been preferred, to allow a smoother user experience.

Textures

For the ground, walls and the trees, textures have been downloaded from the internet. Instead for the finish line and the front texture of the ground, texture have been created. The tree leaves have been created using a bump texture. The result is not really good looking, since it is not in line with the rest of the style of the game, but a different kind of texture was needed for the project.

Initial page and other screens

The initial page is composed of a background image, that is constant through all the html pages, three buttons and the instructions to play the game.
The user need to choose a difficulty that changes the number of enemy robots present in the level, thus modifying the paths and the time needed to complete the game.

To store the choice of the user, the *sessionStorage.setItem()* function has been used, and to retrieve the variable in the javascript files, the *sessionStorage.getItem()* function has been used in the *variables.js* file.

The finish page displays the time needed for the user to complete the game, and a button to play again that redirect to the initial page, thus allowing the choice of a different difficulty if the user wants to.
How the time is calculated will be described in the collision paragraph.

The game over page displays a little message and a button to play again that redirects to the initial page as in the finish page.

The main page displays the game but before it, a loading page is shown to the user, to give the game the time to load all the models and the scene.
The loading page is simply an image, the usual one, modified to have the loading text on it.

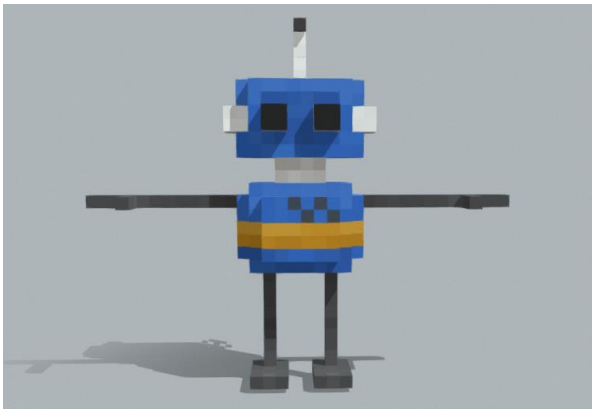
3. Models

External models have been used to create the main player, the enemies and the batteries, downloaded from the internet. To import the models the GLTFLoader library has been used.

In the *init()* function (main.js), after the *load_characters()* function, a timeout is set before creating the scene, to get the time to the models to be loaded. This solution was initially used to prevent fatal errors in the game, before the loading screen implementation. The choice to leave it even after the loading screen, has been made to have a further guarantee.

```
89          load_characters();
90
91          setTimeout(function(){  //wait for all the models to be loaded
92                  initialize_characters();
93
94                  create_level();
95                  battery_animation();
96                  enemy_animation();
97                  input_controls();
98                  inizio = data.getTime();
99                  requestAnimationFrame(animate);
100
101         }, 2000);
102
103 }
104
105 function load_characters() {
106
107         load_robot();
108         load_enemy();
109         load_battery();
```

Robot model

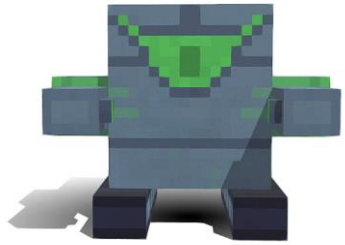


The robot is a hierarchical model, the components used can be seen in the following code:

```
135   const robot_parts = {
136           Root: "body",
137
138           Head: "head",
139
140           Neck: "neck",
141
142           Antenna: "antena",
143           Torso: "torso",
144
145           Rotation: "rotation",
146
147           UpperArmR: "right_arm",
148           UpperArmL: "left_arm",
149           LowerArmR: "right_elbow",
150           LowerArmL: "left_elbow",
151
152           UpperLegR: "right_leg",
153           UpperLegL: "left_leg",
154           LowerLegR: "right_knee",
155           LowerLegL: "left_knee",
156
157           HandR:"right_hand",
158           HandL:"left_hand",
159
160           FootR:"right_foot",
```

Each one of these components has been used in the animations.
The robot model has been used as the player.

Enemy model



The enemy model is not a hierarchical model. To create multiple instances of the model an array has been created, that contains copies of the original model.

Battery model

Also, the batteries have been managed using an array.

Walls and trees have been created using the Three.js *BoxGeometry()* class and applying textures to them.

4. Animations

To create animations in the game the Tween.js library has been used.
The following animations have been created.

Run

It is the main animation, it allows the player to be moved. It is composed of two tweens chained together to create the continuous back and forth movement of each component, and a third tween that restores the initial values (the idle position) when the input keys are left.

In these tweens angles of the hierarchical model of the robot are modified and also the robot position and the camera position are upgraded following the input of the user (a simple check is performed to derive the right value for the step variable that determines the direction movement).
Four dictionary of values are defined: the start, middle, end and stop.
The start values are the ones from which the animation start and that are upgraded by the tween functions. The middle and the end values are the final values of the two tweens chained together, that allows to have the back and forth movement of the components, so for example the arms have a certain value in the middle dictionary and the opposite value in the end dictionary, so it will swing in a direction in the first tween and in the opposite direction in the second tween.
The stop values are the ones that define di idle position.
The *yoyo(true)* and the *repeat(Infinity)* methods have been used for the chained tweens to repeat the animation until the input keys are pressed.
In this animation also a collision check is performed, described later.

Rotate

This animation simply allows a smooth rotation of the player when the direction of the movement change.
A check is performed to figure out the correct rotation angle. The algorithm implemented make use of different auxiliary variables to make the rotation in the right direction. (It happened in the tests that instead of rotate to the right of 90 degrees the model rotated left of 270 degree).
The function also restores the rotation of the collision box of the model, in case strange rotation, resulting from collisions, happens.

Battery animation

Is simply a up and down animation.

Enemy animation

It uses 4 tween functions chained together and so 4 dictionaries of values to create the effect of moving in “circle”. A randomized movement has been tested initially, but it was too complicated to handle due to the collision bouncing problem described later.

Wave

Is the animation executed when the player reaches the finish line.

The animation rotate the player in the direction of the camera and starts a back and forth movement of the arms, intended as an exultation.
The animation is repeated until the finish page is displayed.

Collisions

For the collisions the Physijs library has been used.
Each component of the scene has a Physijs BoxMesh associated, so that collisions could be verified.
The only exception is the robot that has 2 boxes, used to prevent it from falling as result of a collision.
A strange bouncing effect has given a lot of problems, and the use of 2 different boxes for the user is just one remedy to this problem.
The lower box is used for all collisions.

Collision with the batteries, make them disappear and make the user go faster.
Collision with the enemy robots make the game end with a game over.
Collisions with the walls and trees are handled by the run animation that thanks to the *collided* flag and the collision_direction variable, prevents the bouncing effect and the intersection with the walls.
Collision with the final line is done thru a transparent box. When this collision happens, the time to finish the game is calculated, the input from keyboard is disabled, the wave animation starts and the orbit camera is enabled. After 5 seconds the finish page is displayed.

In the *init()* function, after the timeout, a time is saved in the *inizio* variable.

```
98                  inizio = data.getTime();
```

When the collision with the final line happens, another time is saved and the final time is calculated.

```
302                 if(other_object.name == 'finish'){
303
304                         fine = new Date().getTime();
305                         tempo = (fine - inizio)/1000;
```

After the time value is passe thru the *sessionStorage.setItem()* function to the finish page.

Input

The user can interact with the game using both the directional arrows and the WASD keys.
A series of checks and flags are used to determine the direction, to handle the collisions in the run animation, to start the animation, to continue the animation when the key is not released and stop the animation both when the keys are pressed and when the keys are released.

5. Final thoughts

The game function in both Firefox and Chrome browsers, where it has been tested. The game has been developed using Firefox, so it is the most reliable browser.
As already mentioned, some problems with collisions happened during the development, some have been margined, but when a collision happens with the angle of a block (that is a wall or an enemy) the model may rotate and create a strange movement effect, but as soon as a key is pressed the rotation will be fixed. Once it happened that the models where not loaded yet when the game started, but it should be a problem related to internet connection.

6. References

Three.js: https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene
Physijs: http://chandlerprall.github.io/Physijs/
Tweenjs: http://tweenjs.github.io/tween.js/
Robot: https://sketchfab.com/3d-models/funky-robot-180b02d92eea4d65a383d7646c54a3c0
Battery: https://sketchfab.com/3d-models/3d-pixel-battery-f09cf5f373d44e51b5555e133106d7e0
Enemy: https://sketchfab.com/3d-models/3d-pixel-industrial-transport-robot-28beefdfe56041ae96948c6109ebec0b