INTERACTIVE GRAPHICS

N46 THE EXPLORER

Student: Francesco Frattolillo
Student ID: 1783651
Sapienza University
Master in Artificial Intelligence and Robotics

INTRODUCTION

N46 The Explorer is a simple game that embeds all the main topics that we have seen through the course of Interactive Graphics. The game has been created through the *Three.js* library. The aim of the game is to move our robot and reach the second room avoiding the guard, after we have reached the second room we have to destroy a cube through the attack action. If the guard discovers us, a *game-over* modal window will appear and we are able to play again. In the same way, if we reach the cube and destroy it a *You Win* modal window will appear and we can start the level again.

HIERARCHY

N46 is a robot built only through Three.js geometry objects. The structure is the following:

- **Cube**: this is the main object and it contains all the others. The idea is to use that cube for collision check that will be explained later.
- **Head**: it is just a sphere, but it's total angle is 180 degree instead of 360.
- **Upper Body**: the upper body is represented by a cube and it is connected to the two arms and the lower body.
- **Arms**: both the arms are composed by three spheres representing the shoulder,the elbow and the hand. These three elements are linked through two cylinder.
- **Lower Body**: lower body has two pieces, a cylinder with small heigh and an inverted cone. Instead of rotate the cone, we just set its heigh with a negative value in order to not change it's local axis.
- **Legs**: legs are composed by a joint on top (a sphere) and 2 cylinder. One cylinder is almost inside the other, and this is done in order to make a more realistic jump animation.
- **Wheel**: the robot has a single wheel that is connected to the two legs and it's implemented through a rotated cylinder.

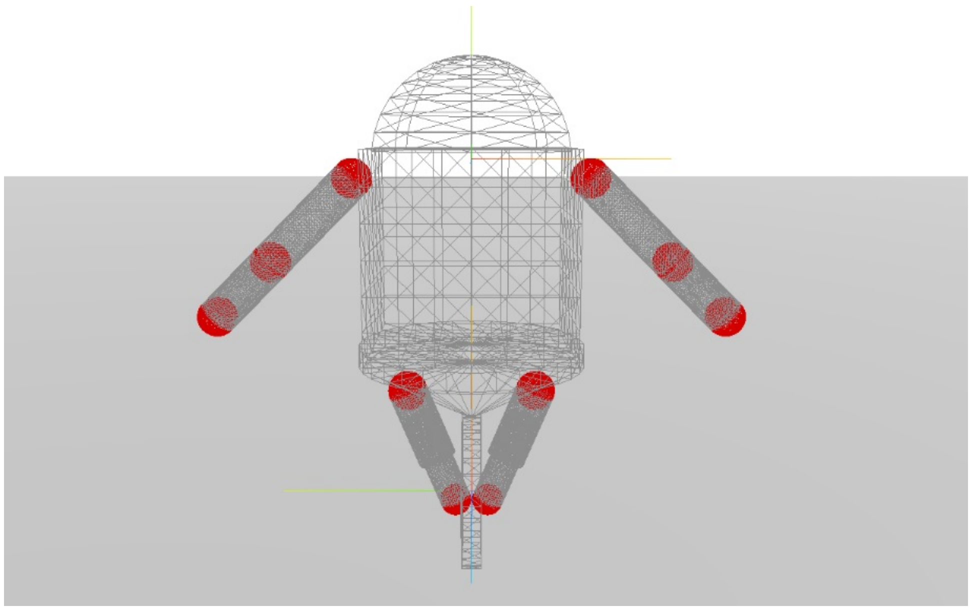

Figure 1: N46 Model

LIGHTS AND TEXTURES

In the scene there are two differents kind of main *light* sources, it is possible to switch between them through the control box in the top right corner of the game page. By default we have a directional light in position <100,100,100>. The other elements in the scene are:

- **Floor**: floor is obtained through a plane geometry and its attributes are set in order to receive shadow but to not cast them.
- **Walls**: we have a set of walls (cube geometry) shown in the figure below.
- **Door**: the door is placed between walls 13 and 14, it is composed by two cubes and the idea is to emulate a sliding door that will open when the robot is near it.

There is a function that check collision between robot and all the walls and in case of collision the robot will be pushed back.

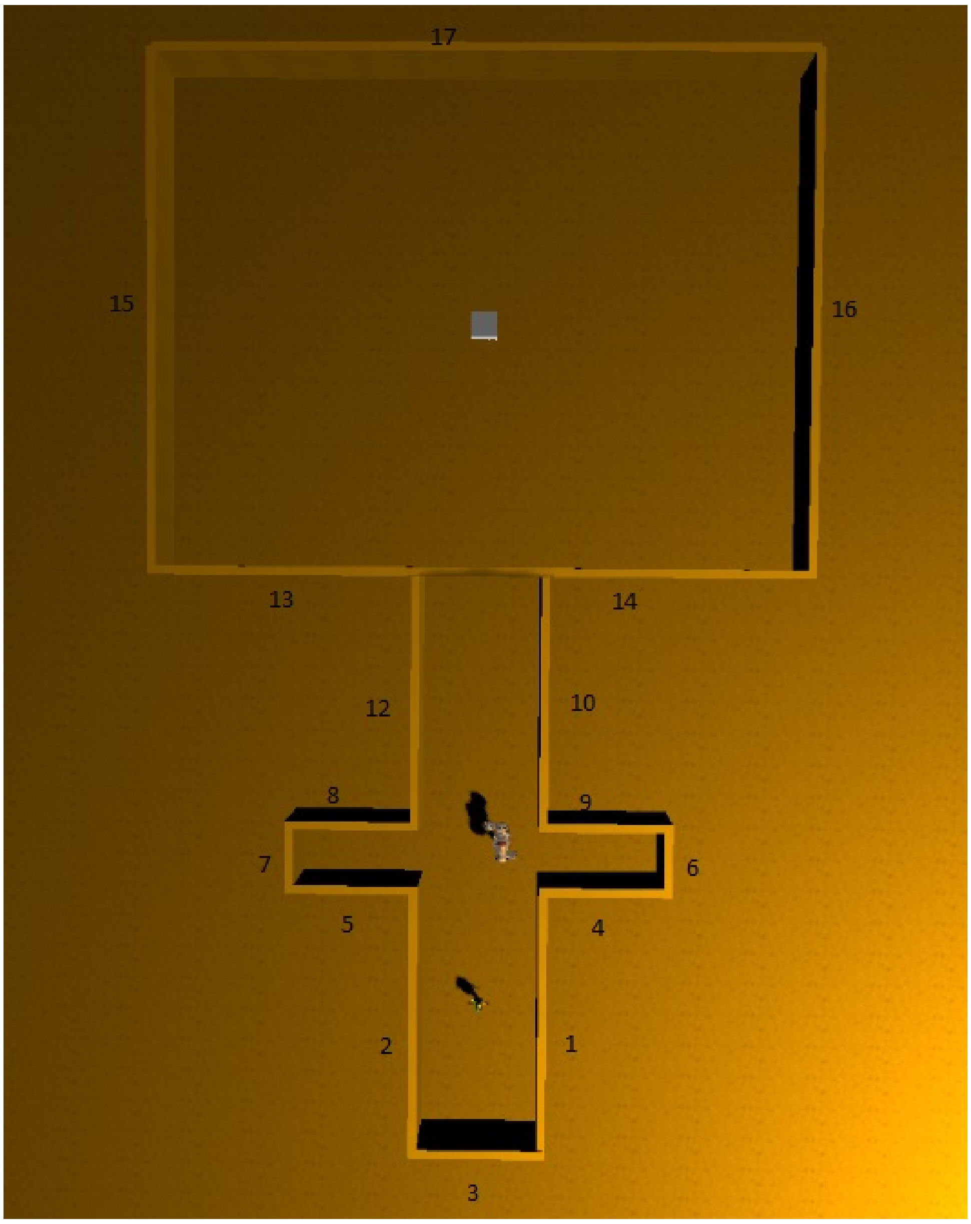

Figure 2: Walls position and number

As said before, we can switch from directional to point lights. There are 5 point lights in the environment and they are positioned in the middle of walls number 1, 2, 10, 12, 17. Initially these types of lights had been set with the ability to cast shadows, but this was an extremely expensive operation, in fact all the actions inside the game were slowed down. To avoid this problem we turn off this option, actually only the directional light is able to cast shadows. Textures are applied to the floor, the walls and the sliding doors, while the robot has no texture. The metallic effect of the robot is obtained using the *MeshPhysicalMaterial* class.

User Interaction

In order to make the game a little bit more interactive, we have added a control box in the top right corner that allows the player to change some element inside the scene. To create this control box we have used an external javascript file called *dat.gui.min.js*. The element that can be changed inside the scene are the following:

- **Light's Type**: As said before, the user can switch between a directional light and some point lights .
- **Light's Color**: There are 3 different color available and they are orange, white and light blue.
- **Camera**: The user can switch between a camera behind the robot, a camera on top of the robot and a free camera that allows the user to freely move in the environment through the mouse.

In order to allows the free camera to be controlled by the mouse, we have used a javascript file called *TrackballControls.js* . The switch between the different cameras and lights is obtained using global boolean variable; based on the value of these global variable we are able to render the scene using one of the three cameras and in the same way we are able to decide which light to display.

Animation

There are two animated objects inside this project. The first and easiest one is the guard, it is a 3D object loaded inside the program using the *GLTFLoader* module. The guard model also has a walk animation. In order to make it work, we have added a *Three.clock* that allows us to take the next frame inside the Animation Clip. This is not enough to make the guard move, hence we have added a function that allow the guard to translate along its x-axes. The guard movement is limited and when the guard reaches that limit it will turn back and inverts its movement's direction. The second animated object is the robot N46. On the first attempt the idea was to control the robot through the *document.addEventListener* function, but the problem was that the movements were really mechanics when more than one key were pressed or released at the same time. Hence we used the javascript module *threex.keyboardstate* in order to have smoother movements. Other animation as *jump* and *attack* were initially implemented through the *setInterval(f,T)* function; this function allows us to call another function f every T milliseconds. The problem in this case was that sometimes the actions didn't stop after the end of the function. Hence we decided to choose another way, we set three boolean global variables *notJump, notJumpAhead, notAttack* that allows us to execute the desired action without being interrupted from another keyboard press event. All the actions uses an internal counter that is incremented everytime the frame is refreshed. Based on this counter, some parts of the robot will be moved:

- **Jump**: this animation will move the cube around the robot and all its children along the robot local positive y axes. At the same time the joint's shoulders rotates and the lower leg will slide down as a piston, the wheel will move down too in order to not lose conctact with the lower leg (the wheel is a child of torso hence it will not automatically move down with the leg).
- **JumpAhead**: jumpahead is a mixed function between walk and jump, in fact the robot will continue to move along its local z axes while jump.
- **attack**: this function allows the robot to raise its arms in an horizontal position and to rotate by 360 degree around its lower torso. In the end it will return to its initial position.

Inside all of these actions are implemented the *checkCollision, openDoorfun* and *moveG-uardBB* functions in order to let all the elements in the scene to act in the normal way.

COLLISION

The collisions are implemented through the *Raycaster* class. The idea is to check if the rays starting from the center of the robot and passing through it's vertices will intersect the face of the other objects. In the image below we can see the different bounding boxes used to this purpose (they are still present in the final game version but they are made invisible through the object material property):

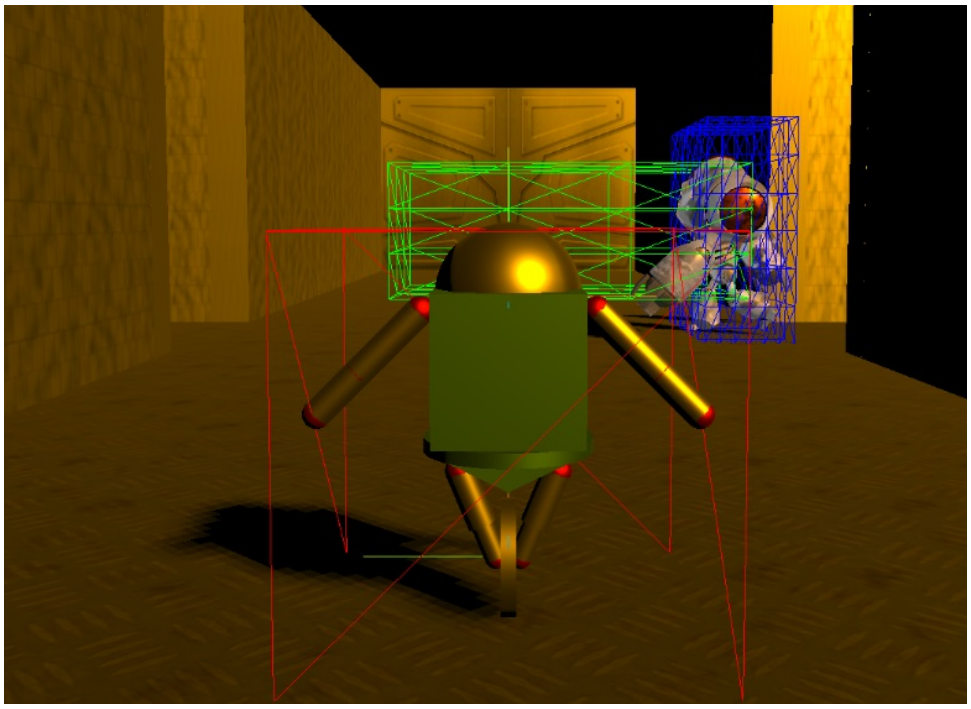

Figure 3: Bounding boxes of N46 and guard

As we can see the guard has two bounding box, one is used for check collision and the other one is used to emulate the guard's sense. In the first attempt we tried to use a global checkCollision function, that will check collision between robot and every wall in the scene. Obviously this operation was really expensive, then we decide to change and check collision only between element in the same area. This is obtained just by a simple check of the robot global position. The idea behind the checkCollision function is that if the robot collide with a given wall, it will be pushed back by a certain amount that is higher of its speed otherwise the robot would overtake the walls.

Other Functions

Other function used in this project are:

- **update**: Here we check if a keyboard has been pressed and based on this outcome we will execute the relative animation action instead of this function.
- **animate**: This is the main function that allows us to render all the scene. Based on the values of the global boolean variables we choose which camera between the three that had been show before will be rendered. In this function we also decide which

action to execute and we block the execution of the other ones. In this function we also increment the value of the variables *increment* that is used by the *explode* function.

- **explode**: the goal of this function is to simulate an explosion when the action *attack* is pressed near the cube in the second room. This is achieved by storing all the small polygon(triangles) that compose the cube's mesh and pushing them away when the function is activated. We can see the effect in the immage below
- **gameover** : this function will stop the current rendering and show a modal windows with a *GAME OVER* text and a button used to restart the game
- **win**: this function is activated exactly when the explosion function ends. As for the gameover function, it will stop the current rendering and show a modal windows with a *YOU WIN!!!* text and a button used to restart the game

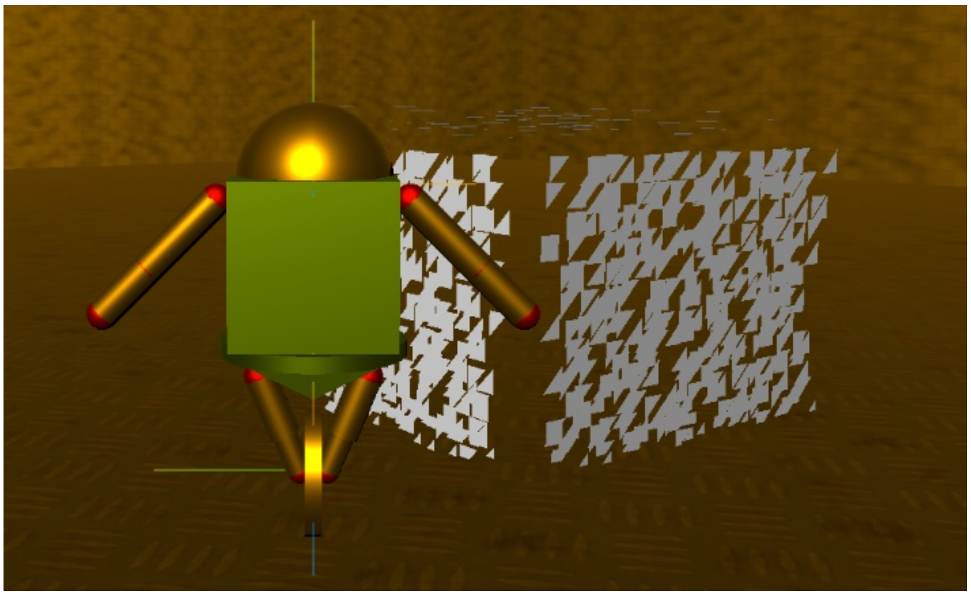

Figure 4: Cube explosion

LIST OF LIBRAY USED

The list files and library used are the following:

- **three.js**
- **three.min.js**
- **TrackballControls.js**
- **threex.KeyboardState.js**
- **GLTFLoader.js**
- **dat.gui.min.js**
- **info.js**