INTERACTIVE GRAPHICS PROJECT

REPORT

Authors:

Lorenzo Guercio – 1913660
Kevin Munda – 1905663
Lorenzo De Rebotti – 1745942

Università degli studi “La Sapienza”, anno 2019/2020

Index

1. Introduction

The project idea consists in a small 3rd person shooting videogame –"The dark saviour"- divided into three levels, where the main character is a robot that has to deal with an increasing number of enemies - soldiers - and has to defeat them in a limited period of time. Initially, the player can choose from three different degrees of difficulty, which determine the strength of the enemies and the speed of their bullets. After the start of the match and after the completion of each level, a countdown is set and the user has to kill each enemy before the countdown goes to zero: the player loses if the enemies kill the main character or if the time expires.
Two additional elements are added to the environment: a power-up, responsible of increasing the life of the player, and a shield, responsible of momentarily protecting the player from the enemies; these two additions spawn casually in the environment every short period of time. These two elements can be collected by the player simply passing over their position in the ground.

2. Pre-scene

The initial scene is set alongside with the camera and the renderer; it is simply composed by three PlaneGeometries indicating the degree of difficulty that we want to face during the game. These geometries are covered with a texture and oscillate periodically in front of the camera. At the top left, the commands of the game are highlighted, while at the top right the icon of the background music is present. The "Start" button below must be pressed in order to start the match after having chosen the difficulty: with this action, the necessary HTML elements are hidden and the new HTML and Three.js elements needed are added to the scene.

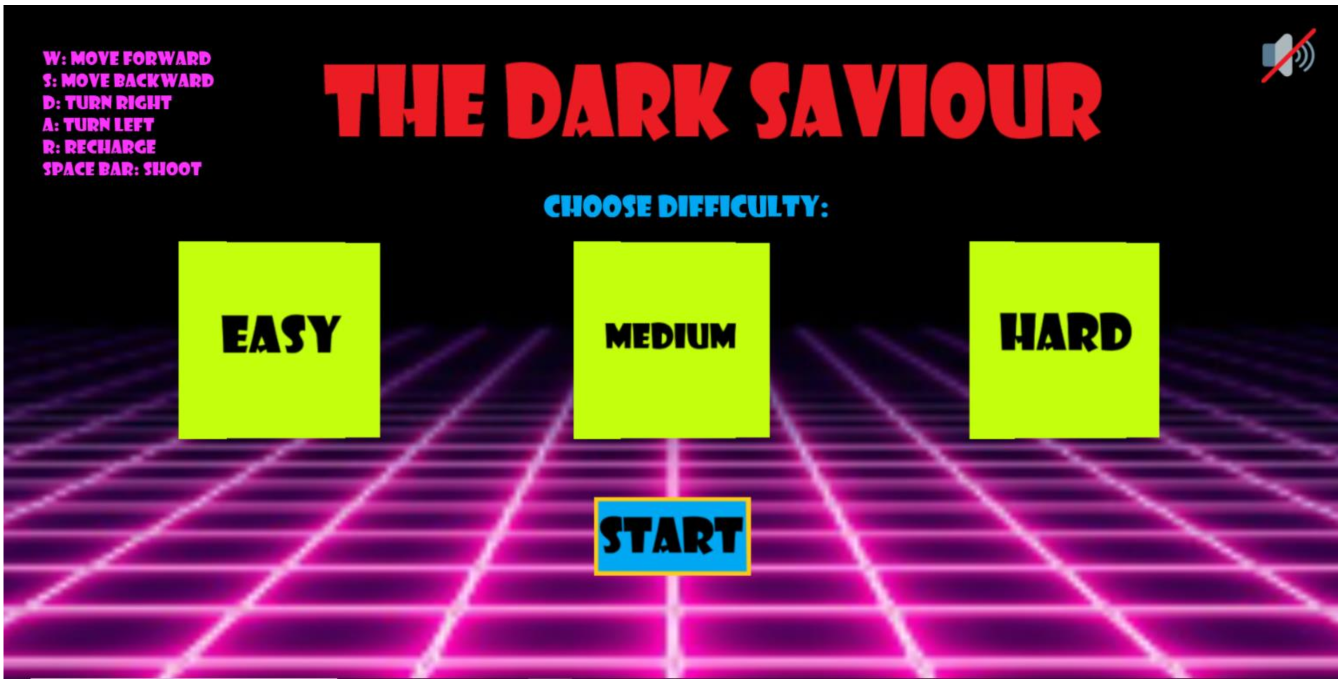
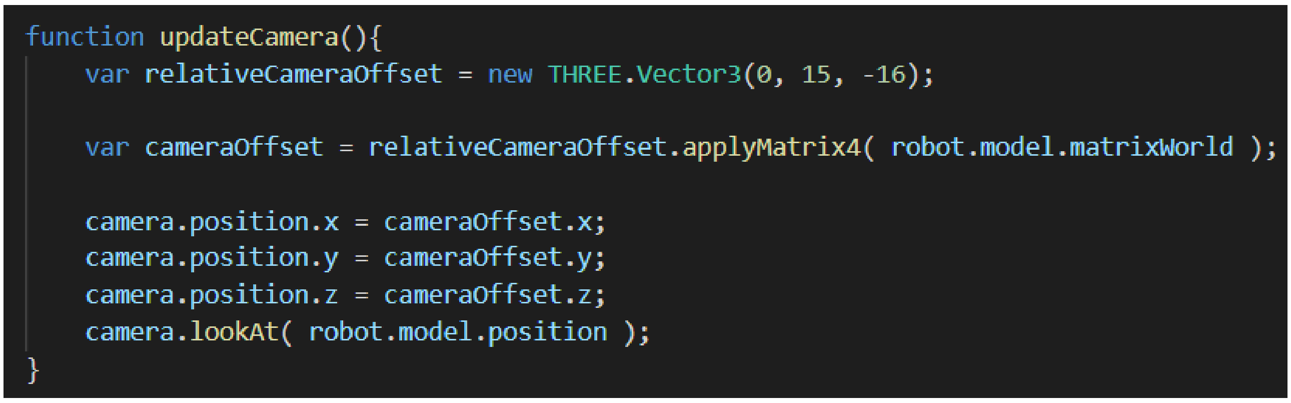


Figure 1: initial scene of the game.

3. Main scene

The main scene is composed of several Three.js elements: the ground is composed by an initial zone, a corridor and the central arena, that becomes bigger in reference to the level and to the number of enemies in the environment. Each zone is a THREE.Mesh composed of a PlaneGeometry and a MeshPhongMaterial, each with different coordinates.
The walls wrap up the ground and identify the range of action of the player and of the enemies; they are Meshes composed by a BoxGeometry and a MeshPhongMaterial.
The camera is oriented following the orientation of the player: a simple script is responsible of keeping a constant offset between the camera and the player, with the resulting effect of having the back of the robot always frontal to the camera.



```
function updateCamera(){
    var relativeCameraOffset = new THREE.Vector3(0, 15, -16);

    var cameraOffset = relativeCameraOffset.applyMatrix4( robot.model.matrixWorld );

    camera.position.x = cameraOffset.x;
    camera.position.y = cameraOffset.y;
    camera.position.z = cameraOffset.z;
    camera.lookAt( robot.model.position );
}
```

Figure 2: the code responsible for the camera always following the orientation of the robot.

Then, we have the main elements, the robot and the soldiers, which will be treated separately afterwards in the next chapters. Linked to these models, we also have the bullets, of different type based on the character that shot them.
Additionally, the ‘heart’ and the ‘box_shield’ elements are present: the ‘heart’, responsible of adding some life to the player, is a Mesh composed of an ExtrudeBufferGeometry and a MeshLambertMaterial, while the ‘box_shield’, responsible of protecting the robot from an enemy bullet, is a Mesh composed of a CylinderGeometry and a MeshPhongMaterial.

Talking about the environment, the lighting is very simple: a bright, white ambient light that homogeneously enlightens the scene.
The last element of the scene is the Skybox, a simple way to add a static background in our scene. In our case, the six images composing the skybox form a dark environment where the arena is inserted.
The font used for each written paragraph and specified in the CSS file is ‘Showcard Gothic’.

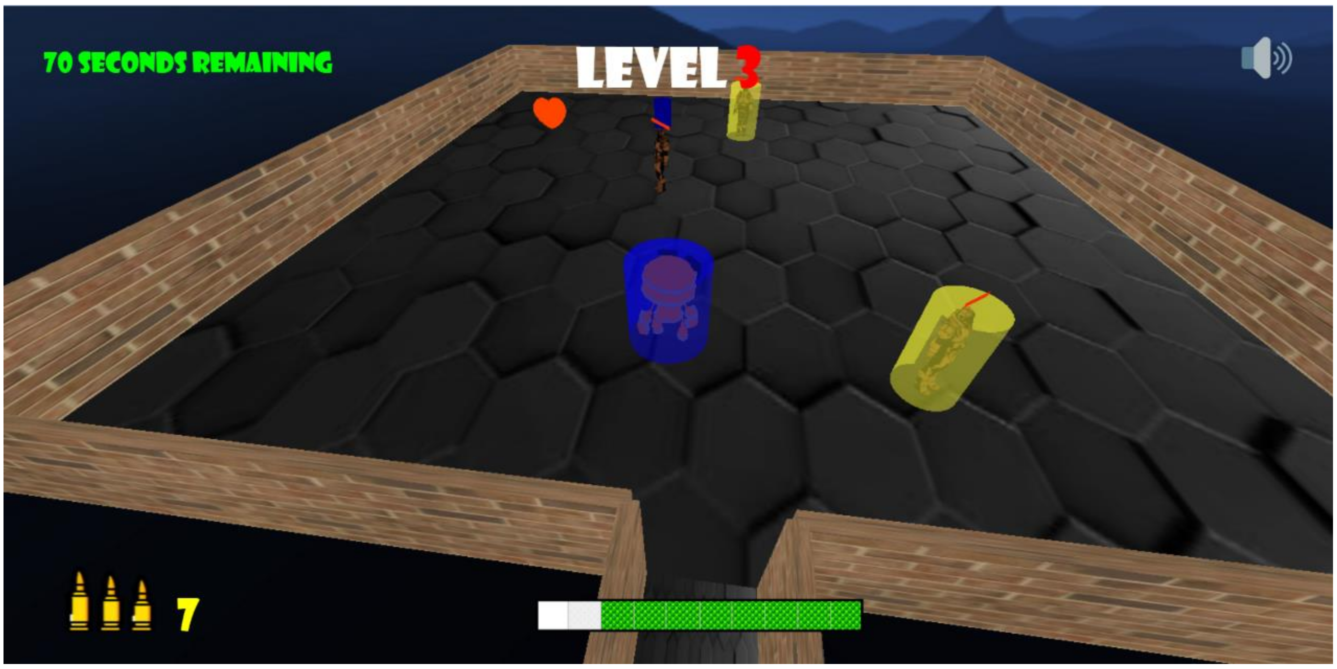


Figure 3: main scene in the middle of a match.

As we can see in the image above, there are some essential HTML elements needed for the correct development of the game: the countdown at the top left reports the time left in the game, the number at the bottom left reports the remaining number of bullets of the player, and the lifebar in the middle decreases its value for each bullet hitting the player.
At the end of the first and second level, an intermission page is inserted: when the user decides to continue and go to the next level, the entire game scene is removed and a new one is built, with only the player's life kept as a parameter. At the end of the game, both in the cases of victory and defeat, there is the possibility to start a new match: in this case, the HTML page is simply refreshed and the user returns to the initial scene.

4. Robot model

The model for the main character of the game is "Robot Expressive"; it was created by Tomás Laulhé, with some modifications by Don McCurdy and it is provided by ThreeJS library, in the 'examples' folder. Although it comes with some predefined animation, they have not been used in this project; all the animations of the robot used here have been realized with TweenJS by us.

The model, that is a .glb file, was loaded via the class "GLTFLoader" which is in the ThreeJS library. Before adding the mesh to the scene, the class "Robot" takes the model as input and it performs some operations:

- Modifications of elements of hierarchical model
- Initialization of the Tween
- Initialization of energy sphere used as ammo
- Initialization of the shield
- Initialization of the hitbox

The main function of the class is called *update*(); it takes as input a set of flags that are set in the main function of the program, according to the user's input, the pointer to the scene and the number of bullets to print on screen. Every time a new action is triggered, for example to move or to attack, the function *clearAnimation*() stops all the other animations and then the particular action is started. Instead the rotations, in any direction, are simply a modification of the model's orientation. Finally, if there isn't any action triggered and the *idleFlag* is true, it will start the idle animation infinitely, until another action stops it.

4.1 Elements of hierarchical model

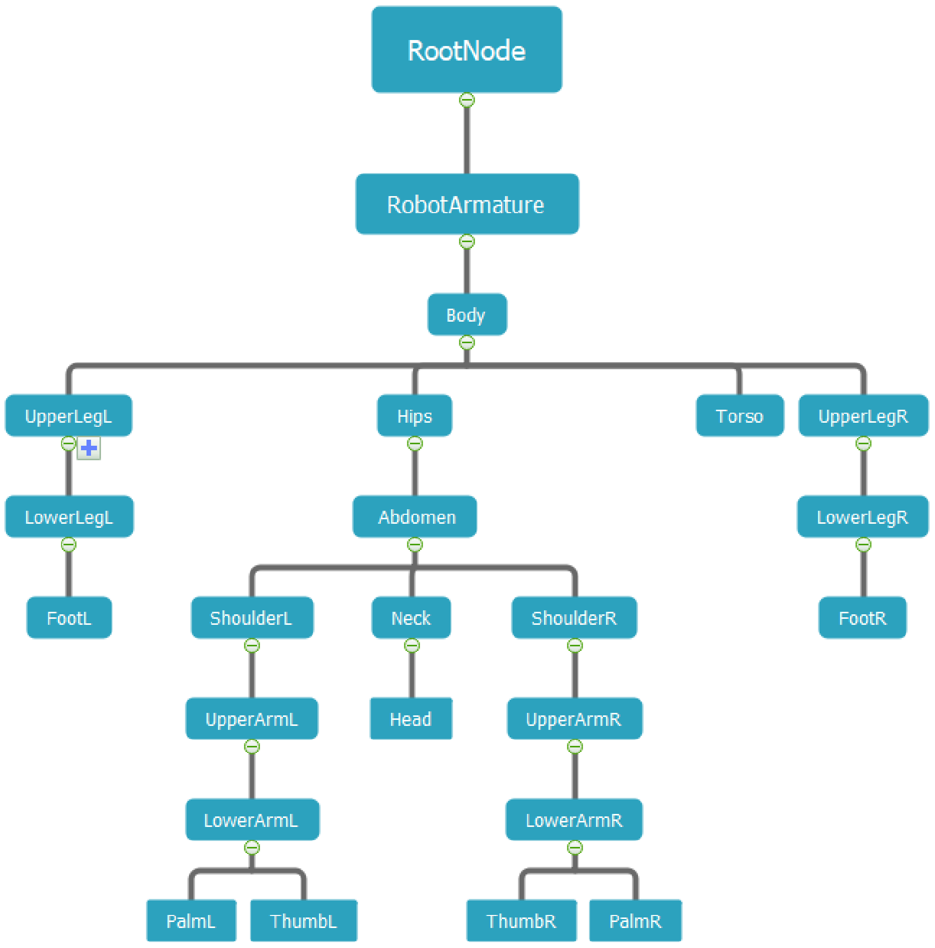


Figure 4: hierarchical model of the robot.

In the figure above, it is possible to see the graph representation of the robot's model. This is not the original one, because of some choice made by the author, for example to put the feet as RobotArmature children and not next to LowerLeg. So, in the *init*() function we modify the structure and we scale the model, in order to have a size that is comparable with the enemy's one.

4.2 Animations

This character has four animations:

- Forward walk
- Backward walk
- Idle
- Attack Animation

All the animations were realized with the library TweenJS with a linear interpolation; in fact, all these motions are simple rotations of the joints' legs and arms, combined with a translation of the model.

Forward walk and backward walk animations are similar, the difference is the direction of the motion and the order of leg rotation to reproduce the gait. These animation starts when the key W (forward one) and S (backward one) are pressed.

Idle animation consists in the rotation of the head, from -45° to 45° and vice versa, if the robot is not performing any animation neither a key is pressed, then it starts.

Attack animation is triggered by the pression of space bar, the robot rotates the right arm and the ammo starts its own animation that will be described in the following.

4.3 Ammo & Shield

The ammo of the main character is a sphere with a texture applied; in the *init*() function we initialize the geometry, the material and finally the mesh. Before the animation, every time the space bar pression is detected, the function *setAmmo*() adds the sphere to the scene and sets to true the flag which tells the main function of the class, called *update*(), that a bullet has been fired. While this flag is true, the class will detect the intersection of the ammo with other objects.

The shield, in a similar way, is a blue cylinder whose components (geometry and material) are initialized in the *init*() function. It is added to the scene as a robot's child when the player intersects the shield powerup on the ground.

4.4 Hitbox & Collisions detection

In the *init*() function we also instantiate the hitbox of the main character, which for simplicity is a cube and not a perfect match of the model's mesh. This choice for sure guarantees optimal performance, although in more complex games it would have led to poor detection of the damage, given that the player could be also hit by a shot in his proximity.

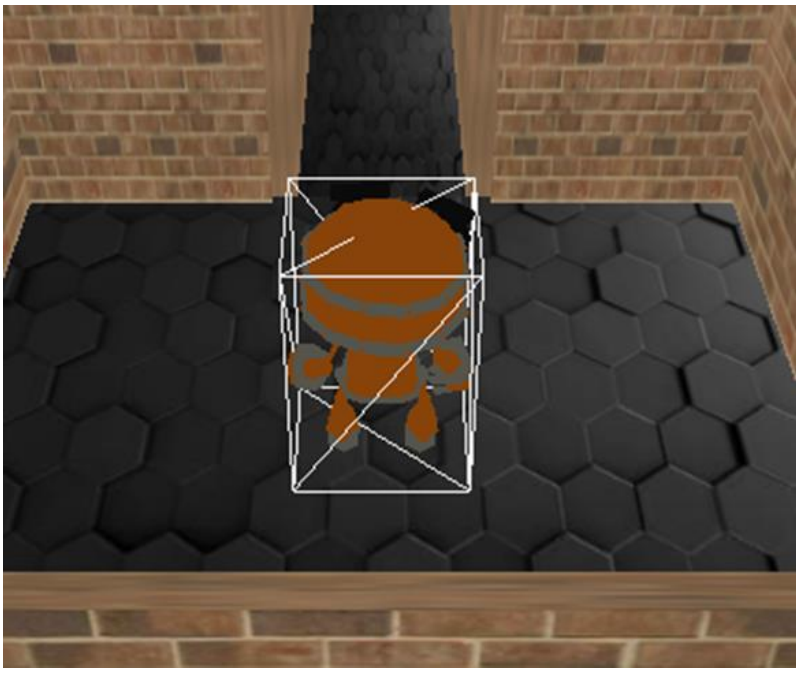


Figure 5: back visual of the robot model and of the hitbox.

The most complex part is the collision detection, that we have done through the Raycaster class provided by ThreeJS library. More precisely, a caster is placed at the bottom of the model to detect the intersection with other objects, while another one is placed at the center of the ammo to detect the intersection with the enemy.

The initialization of the robot's caster happens in the *init*() function, then for every call of the update function the origin and the direction of the caster are updated according to the model pose. After that, thanks to the *detectIntersections*() function, all the objects intersected in front of the model or at its back, changing the direction of the caster, are stored in a vector. For each of these elements:

- If it's a wall or an enemy, the robot stops his motion.
- If it's a heart, the life is restored (10 points, with a max of 100).
- If it's a shield powerup, the shield mesh previously initialized is added to the model and remains in the scene until a damage is detected. If this intersection is detected while a shield is already on, nothing happens.

For the ammo, as said before, the caster is initially placed at the center of the sphere and on a direction that corresponds to the orientation of the model at the moment of the shot. Then, for every call of the *checkAmmoIntersection*() function, the ammo moves forward with linear velocity.

Also this time, with the *detectIntersections*() function, all the objects intersected in the direction of motion are stored in a vector. These elements can be:

- A wall, in this case the ammo is removed from the scene.
- An enemy, in this case the ammo is removed from the scene, after adding the damage to the specific enemy.

5. Soldiers model

The model of the soldier is taken from the set of models provided by threejs.org. The *Soldier*() class contains all the properties, variables and methods that control the behaviour of the model in the environment. The model is loaded in the *init.js* file through the functions *promiseModel*() and *loadModels*(). The latter creates a number of soldiers equal to *numSoldiers*, number that depends on the level of the game, and the parameters and the position of each enemy are set. The spawn position in the arena is randomly chosen by the function *setPosition*().

5.1 Parameters that change with the difficulty

The choice of the difficulty at the beginning determines the values of some parameters of the soldiers. In particular, those parameters are two: the amount of life and the speed with which the energy balls are fired. The function that assigns those values is *setParameters(gameDifficulty)*: when the difficulty is *easy*, the soldier's health is 100 and the speed of the bullet is proportional to 1 (the robot must hit 5 times the soldier to kill him); when the difficulty is *medium*, the soldier's health is 120 and the speed of the bullet is proportional to 1.5 (the robot must hit 6 times the soldier to kill him); when the difficulty is *high*, the soldier's health is 160 and the speed of the bullet is proportional to 2 (the robot must hit 8 times the soldier to kill him).

5.2 Raycaster

The model is provided of a raycaster to check for collisions with walls, other soldiers and the robot. The origin of the raycaster is in the head and the direction is in front of itself. In the constructor are defined the values of *near* and *far*, which determine the range inside which collisions are detected. The collisions are managed by the function *checkSoldierCollisions*(). In this function there are three variables that collect the collision with objects in the environment through the functions of the raycaster *intersectObject*() / *intersectObjects*(). The variable *intersects1* checks for intersections with the children of the scene, the variable *intersects2* checks for intersections with the hitbox of the other soldiers (stored in the variable *arrayOfSoldierMeshesToDetect*), the variable *intersects3* checks for the intersection with the hitbox of the robot (stored in the variable *this.playerMesh*). Then, the data of this variables are stored in one single variable *intersects*. When *intersects.length* is equal to 0, this means that the raycaster did not intersect with any of the objects in the scene; when *intersects.length* is greater than 0, this means that the raycaster did intersect with something in the scene. Depending on which object was detected the soldier can behave in different ways, but this will be defined in the section describing the A.I. of the soldier.

5.3 Hierarchical Structure

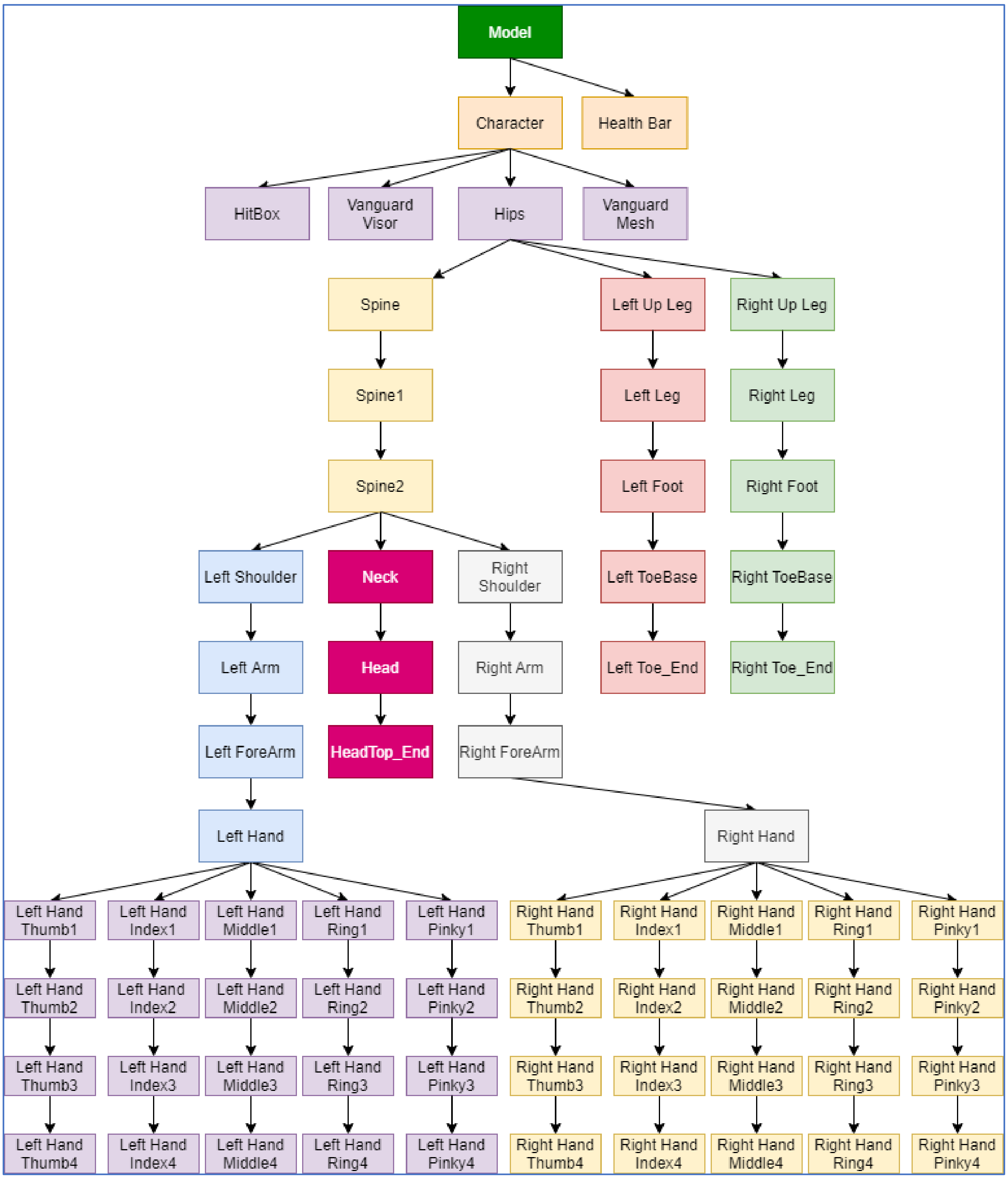


Figure 6: hierarchical structure of the soldier's model.

There are two new components added to the original model: the *Hitbox* and the *Health Bar*.

The *Hitbox* is realized with a Three.js mesh with a cube geometry designed to cover the whole body of the soldier. Its purpose is to verify when a sphere shot by the robot hits the soldier and to avoid collisions with other soldiers.

The *Health Bar* is realized with a Three.js mesh with a plane geometry; it's placed over the head of the soldier and its width changes when the soldier is hit by the robot. It vanishes when the life of the soldier reaches zero. This operation is managed by the function *updateHealth*(). Here, we have an image with the hitbox and the health bar (this is only to show the geometry of the hitbox, because in the game it is invisible):

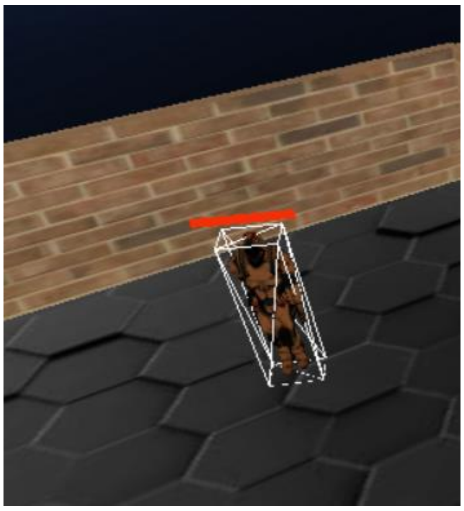


Figure 7: hitbox and heath bar

5.4 A.I. of the soldier

The behaviour of the soldier is controlled by a set of flags that define the current state of the soldier and the transition from one state to another. The function *checkFlags()* controls these operations, assigning the right values to the flags and triggering the right functions for each possible condition. All the animations of the model are done by hand using the library Tween.js.

The possible states of the soldier are:

- ***Idling***
- ***Patrolling***
- ***Aiming***
- ***Shooting***
- ***Reaching the Player***
- ***Dying***

5.4.1 Idling

Each soldier starts from this state. The function that controls the operations in this condition is *idle()*: here the animation restores the original positions of the arms and legs of the model, such that at the end the soldier is standing with straight legs and the arms along the hips. This state can be reached from different situations: when the soldier ends its patrolling, when he finds a wall or another soldier in front of him (through the intersections detected by the raycaster), or when the robot hits him. In particular, in the latter case, he enters in the idling state and when the animation is over, it starts to search for the player. If, after he has reached the previous position occupied by the robot (the position from where the robot has shot), he doesn't find it, the soldier returns in the idling state. An important feature that the soldier has in this state is the shield. In fact, when he stands still, a shield covers the whole model and protects him by robot's attacks, so that in this interval the soldier cannot be damaged. The shield is realized with a Three.js mesh with a cylinder geometry. When the soldier is not anymore in the idling state, and in particular when the new target point to reach is defined (operation that is done in the *patrol*() function or in the *reachPlayer*() function), the shield disappears.

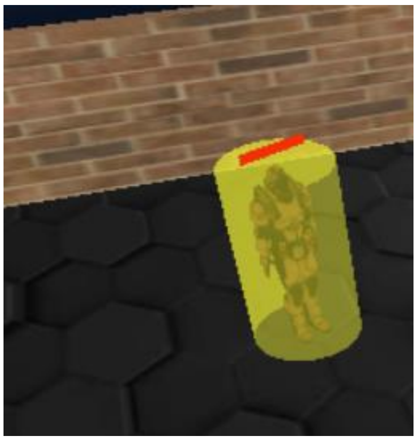


Figure 8: while idling, the shield appears.

5.4.2 Patrolling

In this state the soldier starts to patrol the environment. This operation is managed by the function *patrol*(): at the beginning a new target point to reach inside the arena is defined randomly; when this is done, the soldier rotates towards the new point (animation done with Tween.js) and when the rotation is completed, he starts to walk. In particular, in the function *patrol*(), the animations implemented are the *rotation* and the *translation*, while the actual walk animation is defined in the function *walk*(): here the upper and lower limbs are rotated so as to simulate a realistic walk.

During this state, the soldier has no shield, so he can be damaged by robot's attacks.

When he reaches the target point, if the robot wasn't found along the path, he returns in the idle state and starts again to patrol.

Figure 9: the soldier walking while patrolling.

5.4.3 Aiming

The soldier enters in this state when the raycaster finds an intersection with the hitbox of the robot. The animation is managed by the function *aim*(): the right arm is raised upwards while the palm of the hand opens, parrying perpendicular to the ground. The animation continues and can be completed only if the robot is detected throughout the animation interval. If during the animation, the robot moves and so the soldier's raycaster doesn't find anymore an intersection with the robot's hitbox, the aiming animation stops and the robot returns at the idle state. If the aiming animation is completed, this means that the soldier has still the robot in its range and then the soldier enters in the shooting state.

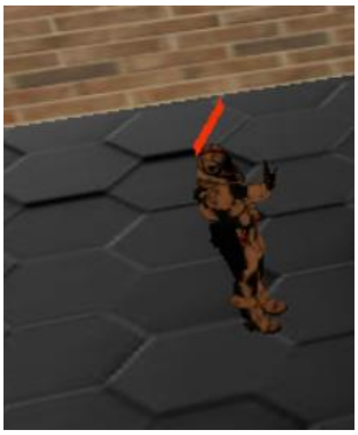


Figure 10: the soldier aiming.

5.4.4 Shooting

If the aiming animation is completed successfully, the soldier enters in the shooting state. In this case the flag *canShootFlag* is set to *true* and this allows to execute the function *createBullet*() (to execute this function the flag *bulletReadyFlag* must be *false*). This function creates the bullet, that is obtained with a Three.js mesh with a sphere geometry to which is applied a texture. Each bullet has a raycaster to check for collisions. The range of this raycaster is very little, to make sure that it finds collisions only when it is very close to the objects. The intersections of the bullet are managed by the function *checkBulletCollisions*(). If the bullet hits the robot, the function of the robot *addDamage*() is triggered.

When the bullet is created, it is ready to be shot. The translation of the bullet is implemented in the function *shootBullet*(), in which is possible to see that the speed of the bullet depends on the parameter *bulletSpeed*, whose value depends on the difficulty of the game. The bullet is "fired" from the palm of the soldier. Each soldier can shoot one bullet at a time; the soldier can shoot again when the previous bullet collides with something in the scene, like a wall or the robot. While the soldier is shooting, he can be damaged and if this happens the shield doesn't appear in this state.

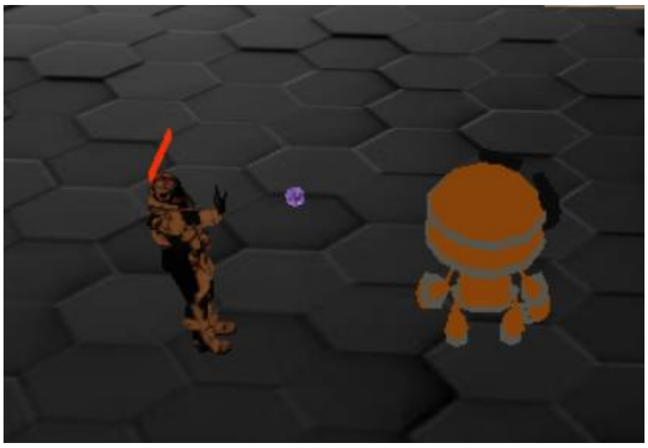


Figure 11: the soldier while shooting.

5.4.5 Reaching the player

The soldier enters in this state when the robot hits him. As said before, when the soldier is hit, first it enters in the idling state. However, this time, at the end of the idle animation, the soldier starts to run towards the position of the robot, in particular the position where the robot was when it shot at the soldier. The function executed in this case is *reachPlayer*() which is similar to the function *patrol*(), the only difference is that in *patrol*() the target point was random, in the function *reachPlayer*() the target point to reach is determined by the position of the robot. The run animation is implemented in the function *run*(); even this function is quite similar to the function *walk*(), the only difference in the two animations is in the time interval in which the tween animations must be executed (in the *run*() function the time interval is 300 ms, in the *walk*() function the time interval is 600 ms).

5.4.6 Dying

The soldier enters in this state when his life reaches 0. When this happens the flag *deadFlag* is set to *true* and this triggers the function *die*(), which simulates the soldier's death before it is removed from the scene: the dead body starts to fall and when it reaches the ground, the soldier is removed. When a soldier dies, the array *soldiers*[] in the *init.js* file is filtered through the function *checkDiedSoldiers*() in order to remove all the dead enemies, which are removed from the scene.

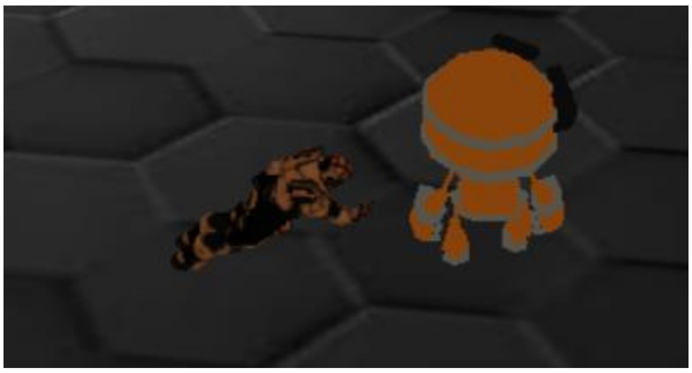


Figure 12: death of the soldier.

6. Possible improvements / bugs

Some improvements could be done to the project: we can see that the performances go down with the increase of the number of objects in the scene: in particular, when there are two or three soldiers in the environment, the bullet speed decreases and the movements of the player become quite slow. Some possible solutions could be to refactor the code in order to make it more efficient and to implement some techniques of asynchronous programming, which can be useful in a Javascript environment.
Moreover, the soldiers occasionally show little bugs in their position and orientation, mostly when they are hit while they are turning around or when they lose the reference to the robot while shooting. However, this is not a common situation and it usually resolves itself in a few seconds, when the soldier is hit again; it is probably caused by the contemporary 'Truth' value of two different flags relates to two opposite actions.
Another bug occurs when too many input commands are sent at the same time, e.g. moving forward, rotation of the model and shooting; the result is a strange behavior of the robot's head animation, with some unexpected and fast rotation.

7. Conclusions

The experience of developing a simple game like this was satisfying and challenging at the same time. The use of Three.js and Tween.js was in particular quite demanding at the beginning, but the strengths of these libraries were always clearer after some time of practice.
As a last consideration, it could be easily stated that Three.js is one of the most efficient libraries for creating and displaying 3D computer graphics, and that the development and gameplay of simple games using Three.js is optimal both for programmers and players.

8. References

[1] https://threejs.org/ : Three.js website
[2] https://threejsfundamentals.org/ : articles to learn to use Three.js
[3] https://createjs.com/tweenjs/ : getting started with Tween.js
[4] https://www.youtube.com/watch?v=6oFvqLfRnsU&t=2328s : YouTube Three.js tutorial
[5] https://riptutorial.com/three-js/example/17088/ : object picking / raycasting
[6] https://www.youtube.com/watch?v=cp-H_6VODko/ : YouTube Three.js Skybox tutorial
[7] https://stemkoski.github.io/Three.js/Chase-Camera.html/ : Three.js camera orientation