Interactive Graphics Project

Knight Run

Adriano Puglisi, Matricola: 1743285
Vincenzo Colella, Matricola: 1748193
Gabriele Margutti, Matricola: 1759610

July 2021

Contents

1 Introduction

The game that we projected and implemented is a one-player game in which a knight must advance and reach the final trophy in order to complete the level. During the level, he will meet different types of obstacles, including skeletons trying to stop him. It is a platform game inspired to the classic arcade game Prince of Persia.

After the loading, the user will find the first page composed of:

- an upper bar in which the available lives are shown on the left, next to it there is the volume button and on the right there is a reminder of the commands
- a table in which are listed all the commands that is possible to use inside the game :
  - **A - D**: are used to walk to the left or to the right respectively.
  - **SPACE**: is used to jump.
  - **SHIFT**: is used in combination with **A - D** in order to run.
  - **K**: is used to attack using the sword
- a start button to start the game

The goal of the game is to reach the end of the level without losing the three lives given at the start. A life will be lost whenever the character collides with a skeleton, or when he gets hit by a spear. To avoid being killed by the skeleton,it is possible to attack him with the sword; with two successful hits, the skeleton will disappear.

2 Level Building

To build the level we created different functions in which every set of objects is created.

The first, main object of the environment of the game is the brick. It's the platform in which the knight will run, walk and jump to advance and avoid obstacles. In the first function, called createBrick(), we first load the models and scale them to the right dimensions. Then, using a for-loop, we clone every brick and place it into it's right coordinates. Using a variable 'i' we are able to set a distance between the bricks in order to create a platform. Thanks to the function setPlateHB, we can attach a hitBox to every brick created to give it its specific characteristics. The hitBox specifications will be further discussed later. Furthermore, after every addition of a brick on the scene, we increment a loading variable. In this way, we can track how many objects have been created so far and know when the game is ready to start.

A similar reasoning is applied for the creation of the boxes, useful for hiding from the skeletons or for jumping over them. We load the boxes, scale them, and then place clones on the scene. Also, we attach the hitBox to each one of them as before, and increment the loading variable.

The spears are obstacles which should not be touched by the character. By colliding with a set of spears, the user will lose a life. We created them by importing the model, scaling and positioning them. Another hitBox is placed for every spear, along with an increment of the loading function.

Finally, we inserted into the scene different torches to give the setting a medieval look. They were imported as the other objects, scaled, and added with a for-loop. Inside the loop we used different if-else statements in order to place the torches in couples over every brick, cloning them and incrementing the loading variable every time.

All the functions presented above were called once for the first part of the level, and then once again in a very similar function (createGroup2, createBox2, createTorch2, createSpear2) but with an offset of 600 in the y axis and some small changes to make the second part of the level harder. In this way we have two similar parts of the level, with the second one being more challenging for the user.

Regarding the background and the terrain, we implemented them with a function createBackground() and createTerrain(). The background (sky) is a plane geometry with specific dimensions in which we add an image texture. We defined the material as a Mesh Phong Material with a FlatShading effect. The terrain is a box geometry in which we attached a texture of a old-looking brick. As in the background, the texture repeates itself many times. For the material, we used a Mesh Phong Material and mapped it with a classic brick color.

3 Animations and Characters structure

There are two different types of characters inside the game which are both imported models:

- **Paladin**: Is our character which we can move, and has a skeleton structure.
- **Skeletons**: Are our enemies, also have a skeleton structure.

For the Paladin have been defined several functions in order to make it able to move and interact with the environment. The first one is *loadPaladin*, this function initializes the starting position of the paladin.
Then we have the function *walk* that generates the walking animation of the paladin in which it moves both legs and arms, and there is a little movement of the upper spine in order to give rotation to the shoulders.
The function *jump* starts the animation of the jump in which the paladin raises a little bit his arms and bends his knees.
The function *hit* let the paladin perform the movement of a sword attack, raising the arm that carries the sword and rapidly going down.

Every time the player presses a key, the latter will be seen by the event *keyDown* and will be stored inside a dictionary containing all the keys pressed associated with a Boolean value which is true if pressed (**keydown**) and false if it isn't pressed or it is released (**keyup**). Thanks to this array it is possible to combine multiple keys to generate different output using the same functions. An example is the "run" command which is the combination of **SHIFT** + **D** or **SHIFT** + **A**, when this combinations are detected, the paladin's speed in movements and walking are increased giving the effect of a real run instead of a simple walk. This "trick" is possible because all the body movement are multiplied by the speed factor. Every time the user stop walking or stop performing an action such as *walk,jump* it will return at the starting position, for the function *hit* only the upper body will return to the starting position.
For the skeleton we have few functions:

- *walk* function which is identical to the paladin's walk
- *raise hands* function which allows the skeleton to raise his arms at 90 degrees with a rotation of the hands during the movement
- *starting position* function which let the skeleton return to the starting position

The first two functions are activated when the paladin is within the skeleton's range of action and the *starting position* function is activated when the paladin is out of the skeleton range of action.

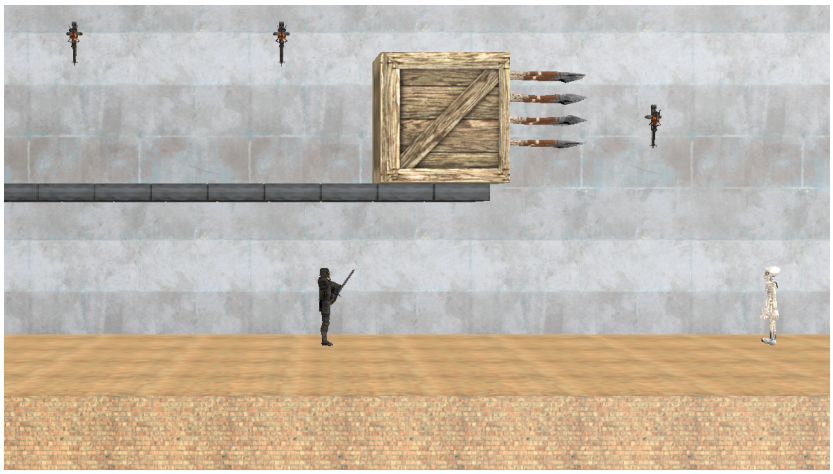
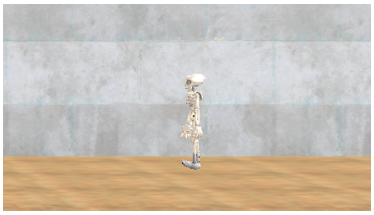


Figure 1: Paladin



(a) skeleton starting position

(b) skeleton moving

Figure 2: Skeleton

4 Physics engine

Physijs is a physics engine used to approximate physical event. It's a plugin for three.js and has an easy to use interface built on top of ammo.js.
In our game it's important to have a physics engine in order to have a more accurate interaction between the objects and make the user feel like playing a more realistic game. The physics engine applies the simple physic interaction between objects and simulate how they would have behaved in the real world. In order to have a perfect representation the computation may become infeasible in a short period of time so we must approximate some calculus to make the process real time.

Physijs allows to set some basic parameters like the gravity and the refresh time, it uses a thread worker to avoid impacting on the performances of the system and has support for all kind of three.js solids and meshes.
Each object has its own friction and restitution that can be specified as its material proprieties. There is a collision detector in order to check when two object

collides, allow to set up the proper collision handler and manage the behaviour of the object after the impact.
In addition, the library provides a way to set the constraint on the entity and allows to decide whether it can move in some direction or can make some rotation over the axis, in general allows to choose which are the degrees of freedom of the objects.

In our game the gravity has been increased form -10 to -30, in order to reduce the air time and have more control over the jumps. In addition the max speed that an object can have has been capped to 5, so the character has some inertia but the sliding after the movement is not accentuated. To have a smooth look over the character movement a mix of velocity increasing and translation has been used, in this way the character seems to walk smoothly over the ground but in fact it is also translating.

5 Hitbox and collisions

To manage the collisions between the objects of the scene we decided to use double rendering. All the objects visible on the screen are actually three.js meshes that are not affected by the physics. In fact there are box meshes created with physijs that are in the same place of the objects but are invisible, in this way we can have simpler management of interactions, because of the simpler form, and still have a good appearance.

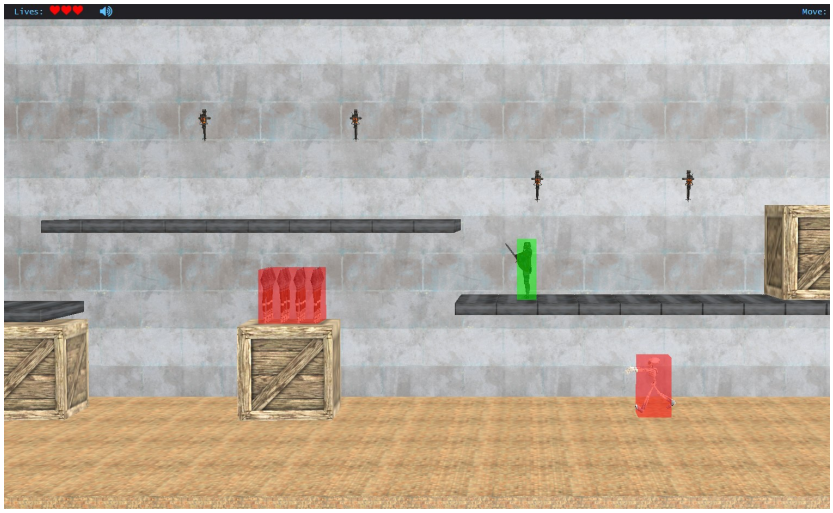


Figure 3: Hitbox and meshes

It has been chosen to use a color code to manage the various kind of collisions, and check whether the main character has been hit by an enemy or is

just standing on the ground. Each inanimate object that is not hurting the player has a blue box of the same dimension of the outside entity. If the player collides with it in the lateral direction, to avoid interpenetration the player will be pushed away from the object.

The entities that hurts the player have a red color. When the character touches them he's translated in the opposite direction of the touched surface to avoid to get hit several times in a row. The character is green and when it touches the enemy that are not in a fixed position, their velocity is set to 0 to avoid the restitution of the force of the impact.
The collision with the ground is used to detect the air time of the player, preventing him to jump if he is in mid air.

To maintain the mesh as simple as possible it has been chosen to implement the player attack not as an object hitting the enemy but computing the distance from the enemy and the direction of the player and in case of the enemy being in range it is pushed back and its life is drained.

6 Game mechanics

All the game mechanics are managed inside the routine of the animation and the rendering. Inside the animate function there is a call to the game routine that manage all the interactions between the player and the enemy, all the changes related to the user interaction, like walking and jumping and the link between the position of the double rendering elements.

Inside the function that handles the game routine we have the handler for the movements of the main character like walking and jumping, the limitation for the max velocity, the detection for the air time.

In addition, there is an opponents AI, that simply computes the distance between the player and each enemy and if they are in range, it start to walk toward the player trying to catch him.

Finally, we implemented the control for the winning condition: the player must have defeated the final boss and reach the trophy at the end of the level.

When the player is touched by some enemy or the spear, he loses a life. If he remains only with one life, the screen flashes in red light and a warning sound is added to the game. When the player loses the final life, the game is ended and the game over scene is displayed with the option to re-run the game.

As introduced before the attack is managed via a the computation of the distance between player and enemy. The actual code of the attack is linked with the swing of the sward, in particular the attack distance is computed when we

have the change off the direction of the sward during the descendent phase.

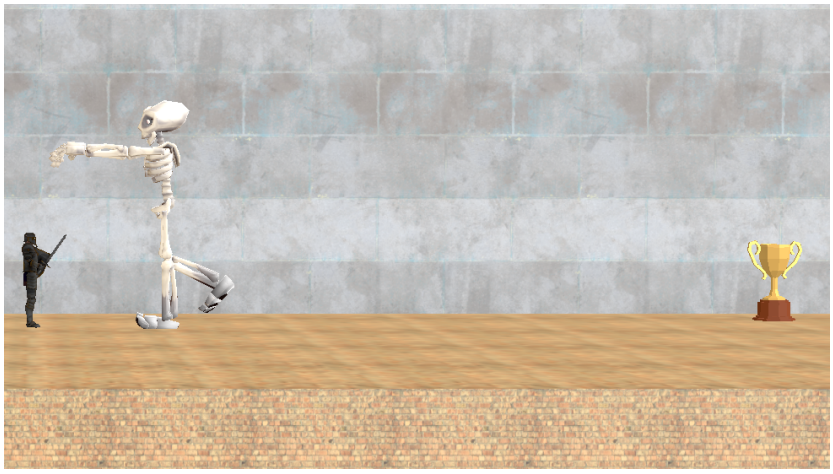


Figure 4: Boss & Cup

7 Sounds and Effects

Inside the game can be found different sounds based on the situation or the action:

- **Background music**: From the beginning and during all the game it is possible to hear a background music
- **Damage**: Is played when the player receives damage
- **Attack**: Is played when the player hits an enemy
- **One Life**: Is played in loop when the player remains with one life
- **Flawless Victory**: Is played if the player wins without loosing lives
- **Victory**: Is played when the player wins with 1 or 2 lives
- **Defeat**: Is played when the player looses the game

Whenever the player receives damage a red ambient light will appear on the screen and then disappear shortly after fading out and when he will remain with only one life, the red light will blink indefinitely.

8 Game

We have a main JavaScript file with the initialization of all the needed structure for the rendering and for the physics engine, then all the entities of the level are created and placed in the right position. The game starts with a loading screen until all the models are loaded along with the physics. At the beginning we have a page where the commands are displayed and when the user is ready can start the game, change to the game screen and enabling the commands.

The UI is very simple: on the top left we have the number of lives and the control for the music, at the top right a recap of the commands is displayed.
If the player loses all his lives, the game is over and the *game over* page will be displayed, while if he manages to defeat the boss and reaches the trophy he will win. In both cases, the option to retry will appear as a button in the center of the screen.

Figure 5: Loading screen

Figure 6: Controls