SAPIENZA
UNIVERSITÀ DI ROMA

ENGINEERING IN COMPUTER SCIENCE

DIAG Man - *Interactive Graphics Final Project*

Submitted by:

Silvia Marchiori *(1475606)*

Giovanni Fiordeponti *(1892763)*

Academic year 2019/2020

Table of Contents

1. Introduction

This relation covers the topic assessed during the development of the *Final Project*. We have decided to realize a *web videogame*, using libraries that exploit the arguments covered during the *Interactive Graphic Course*.

1.1 Idea

DIAG Man is an endless running game inspired by games such as *Temple Run* or *Pepsi Man*, where the character controlled by the player has to avoid obstacles that come through its way, in a scenario constrained by barriers. Since the obstacles are spawned until the player has not lost all his lives (represented by the bag carried by the character), instead of creating an endless field where the player will move, we decided to do the opposite: the player will remain fixed in his position, while all the other objects will come through its way. Meanwhile, the background and the barriers that define the scenario will update the texture to give the motion perception to the player. We can summarize the game through the following features:

- two different *stages* (*City* and *Forest*) with different obstacles to avoid;
- possibility to play the stages on two different *time-days*: *morning* and *sunset*;
- collect coins to get a high-score, saved within the *Session Storage* of the browser;
- move the player from the left to the right using both *the left and right arrow*;
- perform *jumps* using *the up arrow*.

1.2 Libraries

To realize this idea, we decided to use the following libraries:

- *Three.js*: library used to create and display *3D computer graphics* on all *Web Browsers* supported by *WebGL*: it creates a scene in an *HTML canvas object* where different graphical objects can be added. It also has several modules that enable additional functionalities, such as the following ones that have been used in this project:

    - *GLTFLoader*: module that converts *gltf/glb models* into *Object3D instances* that can be added to the scene. Since some of the models used, retrieved from the *Three.js repository* or services like *Sketchfab*, relies on the *Draco library* for decoding their geometry, the loader works internally with an instance of the *DRACOLoader*.
    - *OrbitControls*: module used to fix the camera on a specific target.
    - *Performance Monitor*: it gives the opportunity to show on screen the number of *Frame Per Seconds* (*FPS*).
- *Tween.js*: library used to animate all the various models rendered in the scene. It uses several techniques to move a set of points over the canvas.
- *Physijs*: plugin for *Three.js* that runs a physics simulation on the scene. It offers an easy-to-use *API* to move objects automatically and implement collision systems. It's built on top of the *Ammo physic engine*.
- *SoundJS*: since the game requires some sound to be played during its experience, we have used a library that abstract the *HTML5 Audio Implementation*.
- *Bootstrap*: *CSS Framework* used to design the main menu of the game. It's the only library that is retrieved from the *Internet* by the application and not stored locally.

1.3 Structure

We have decided to divide all the various functionalities of the application as follows:

- *index.js*: the file that contains the main logic of the game and manages the scene where the application will be rendered;
- *model.js*: the file that contains all the functions used to create all the obstacles and the main character;
- *animation.js*: the file that contains all the functions used to start all the animations performed by the models;
- *scenario.js*: the file that contains all the functions to draw the different scenarios and to spawn and update all the different objects in the scene;
- *light.js*: the file that contains all the lights used within the game;

- *constants.js*: the file that contains all the constants used within the application (animation keyframes, speeds of movements and spawning, etc…).

2. Game Logic

In this chapter, we will describe in detail how the main logic of the game has been implemented.

2.1 Moving the Character

As we've hinted before, the character controlled by the player has only three movements: *move to the left*, *move the right* and *jump*. The first two movements are portrayed by translations on the X-Axis, while the second is a translation over the Y-Axis, managed by the *jump animation*. This is possible by adding a listener to the *HTML5 keydow event*, represented in our case by the function "*moveCharacter":*

```
function moveCharacter(keyCode) {
    switch (keyCode) {
        case 37:
            // Left
            if (player.position.x < GAME_BORDER && !isJump) {
                player.position.x += MOVING_SPEED;
                playerBox.position.set(playerBox.position.x + MOVING_SPEED, playerBox.position.y, 0);
                playerBox.__dirtyPosition = true;
                camera.position.set(camera.position.x + MOVING_SPEED, CAMERA_Y, CAMERA_Z);
            }
            break;
        case 39:
            // Right
            if (player.position.x > -GAME_BORDER && !isJump) {
                player.position.x -= MOVING_SPEED;
                playerBox.position.set(playerBox.position.x - MOVING_SPEED, playerBox.position.y, 0);
                playerBox.__dirtyPosition = true;
                camera.position.set(camera.position.x - MOVING_SPEED, CAMERA_Y, CAMERA_Z);
            }
            break;
        case 38:
            //up
            !isJump && jump(camera);
            break;
    }
}
```

Also, the camera attached to the scene has its position been updated by the function, to let the user have a clear vision of the character while moving. The camera, in particular, is a *perspective camera* set behind the player using the *OrbitControls utility*, which forces the camera to orbit around a particular position. Its field of view has been defined as follows:

```
const camera = new THREE.PerspectiveCamera(45, window.innerWidth / window.innerHeight, 1, 10000);
```

The second parameter is the *aspect ratio and* it's related to the window size, that may change during the execution of the script: using an event listener, we can update it and notify those changes to *the camera projection matrix*, which will be updated to reflect the current size of the screen.

2.2 Spawning Objects

While the player is moving through the scene, the objects rendered in the scenario chosen by the user will come forward to him, giving him the impression that his character is moving forward. To automatically let the application manage this process, we have decided to integrate *Physijs* within the *Three.js scene*. First, we have defined the *gravity* applied to the scenario:

```
scene.setGravity(new THREE.Vector3(0, 0, Z_SPEED));
```

In particular, *the constant Z_SPEED* is a negative number, so the object will move from their position to the origin of the scene, where the player is located, through the Z-Axis only. All the meshes that are subjected to the gravity applied to the scene must be shapes implemented within the *Physijs environment*. These meshes come with an additional parameter that needs to be specified, along with *the geometry* and *the material*: the *object mass*, which will tell how much the object rendered to the scene will be moved by the gravity force. In particular, all the objects that the player must avoid will have a non-negative mass, while the player will have a mass equal to zero, to prevent him from being moved by the gravity force. Once all these details are known, we can start the physics simulation by calling the *simulate method* within the *animate function*, which is executed at every frame execution. Meanwhile, the scenario chosen by the player has started a function that will spawn objects to the scene at a given time, using the *setInterval* utility function.

2.3 Collisions

Once the objects are moving properly, we have to find a way to detect whether the player has been hit by any moving obstacle. Thanks to *Physijs*, an object can detect a collision with another object by having a callback attached to the *collision event*

listener defined within the mesh. Of course, both meshes involved in the collision must be shapes defined within the *Physijs library*. After the callback is raised by the mesh, the function *collisionCallback* inside *index.js* does the following:

- it checks whether the object is a coin or not, by checking the name of the mesh (each mesh has been defined with a unique name, which is a combination of the *object type* and the *timestamp* when the object has been created): if yes, it will increment the coin count and remove the coin from the scene, otherwise it will assume that an obstacle has been hit: if an obstacle has a collision animation, it will be started along with a corresponding sound.
- it checks whether the player has any life left. If yes, it will decrement the life count and start the animation to remove the bag, which drops the current bag attached to the player. If the life count goes below zero, the user will receive a game over screen and the collision animation of the player will start.

3. Models

In this chapter, we will describe in detail what type of methods have been used to add different models to the scene.

3.1 Loading Models

We can divide the models used within the game in two different categories:

- **GLTF Models:** models that are loaded from a *GTLF/GLB file*, such as *the main character* and *the bag* carried on its shoulder. In particular, the following models have been used for the *City Scenario*:
    - *a car model*, whose wheels are subjected to a rotation along their X-Axis at each render;*a lamp model*;
    - *a truck model*;

  while for the *Forest Scenario* the following models have been used:
    - *a tree model*;
    - *a gazelle model*.

Each model has an animation (see *Chapter 4*) triggered when a collision with the player happens. When a model is loaded, it must be traversed through all its nodes to let every mesh cast its shadow received from the light to the scene. Since the meshes related to each *Three.js Object* are from the *Three.js library,* it's impossible to change them with meshes from the *Physijs library*. To let these models move according to the gravity force applied to the scene, we have decided, for each model, to attach an *invisible Physijs Box Mesh*. When a model is created, it is stored within a particular array along with the box mesh. The latter will change position according to the scene: the *update function* called within *the animate procedure* will check, with respect to the chosen scenario, the arrays of the models involved and update each model accordingly to its corresponding box. All the invisible boxes can be seen by enabling the *Debug Mode*, by changing the corresponding flag within the *constants.js file*.

- **3D Models:** simple *Physijs meshes* made by *geometry* and a *flat material*, such as the coins that need to be collected by the player. In that particular case, the coin is subjected to a rotation along its *Y-Axis* on every frame. Other models are *the building blocks* appearing in the *City Scenario* and *the rocks* appearing in the *Forest Scenario*: in those two cases, *a texture* is mapped to the material.

3.2 Hierarchical Models

Hierarchical models give the opportunity to manipulate more objects at the time following a given set of constraints. This concept has been used in three different models within the game:

- *the car model*, which consists of *the chassis* (with all its components) and *the wheels*. Wheels are reached through their name within the hierarchy by calling the function *getObjectByName* on the car model: each of them is stored in an array to be used later during the rendering process, to animate them by changing the rotation speed. The car structure is also exploited to animate the car during a collision with the player;
- *the gazelle model*, which consists of a *body* where *legs* (made by two objects, representing the upper and the lower part of the leg) and the *head* are attached.

This structure is exploited in the same way as before to animate the model on two different occasions, the run and the collision;

- *the player*, which is a hierarchical structure made by using the *Group Object*, has two children: the character model (with all its components) and the bag model. Since those two models must move together accordingly, it's easier to manipulate the instance rather than update the movements of both meshes at render. Also, the animations performed by the player have been developed by traversing the children of the *Armature node,* stored within *the skeleton variable*. Also, each time an obstacle has been hit by the player, the bag is removed from the *Group instance* and, after the *dropBag animation*, added again if any life is left.

The hierarchy of each model has been explored by calling for each of them the function *dumpObject* from the *model.js file*. One might ask why all the transparent *Physijs mesh boxes* haven't been attached to their respective model by the usage of a *Group object*: this was impossible since the *Group class* accepts only *3D Objects* from the *Three.js library*.

3.3 Textures

Besides the texture mapped to some of the models rendered in the scene, another set of texture has been added in different parts of the scenario:

- *in the background* of the scene, where also a *fog* has been added. This choice makes the appearance of the obstacles more natural and less sudden;
- *on the right and left barriers* of the scenario, which are the boundaries that limit the movement of the player within the scene. Since they must give the perception to the user that the character is moving forward, they're translated by changing their offset using the *setInterval* function. To do so, the texture must be wrapped both vertically and horizontally by following a *"repeat"* strategy, which means that the texture is repeated an infinite number of times following the same *UV Mapping* adopted by the mesh in which they're in;

- *on the ground* of the scenario, by applying the same intuition as before. While the texture's barrier is repeated only in one direction, the texture related to the ground is repeated along with both *U* and *V* directions.

4. Animations

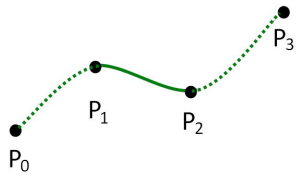
In this chapter, we will describe how the animation system of the game works.

4.1 Overview

To make all the animation (besides the rotations of coins, rocks and car's wheels) we have used *Tween.js*: it's a library that gives the opportunity to perform a movement, called Tween, that may be a rotation or a translation in one or more directions, using different interpolation functions. This movement is constrained between two keyframes, that represents coordinates of a particular mesh in the scene: different sets of keyframes may follow a particular sequence by starting a Tween after the finish of another one by using the *onComplete callback function.* Also, a movement can be repeated a various number of times by defining the number of repetitions with the *repeat function*.

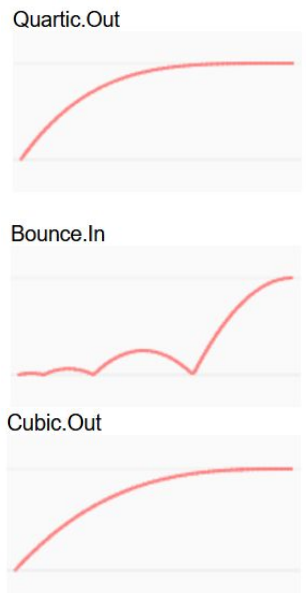
4.2 Character

The character has three animations: *run*, *jump* and *collision*. Most of the movements done within those animations are done by applying *the spline interpolation* between the initial and the final point, rather than applying a basic linear interpolation. It gives to the animation a smoother flow, and it's done by choosing a particular spline curve within *Tween.js* by calling the interpolation function of the *Tween object*: in particular, we have chosen the *Catmull-Rom interpolation* method, which draws a curve between two points by using a recursive algorithm involving four control points. Also, for the jump animation, we have applied an *easing function*. It is a nonlinear function that can be applied to both ends of a *Tween*: in that particular case, we have applied an *Elastic effect* to ease out the lift of the character.



4.3 Obstacles

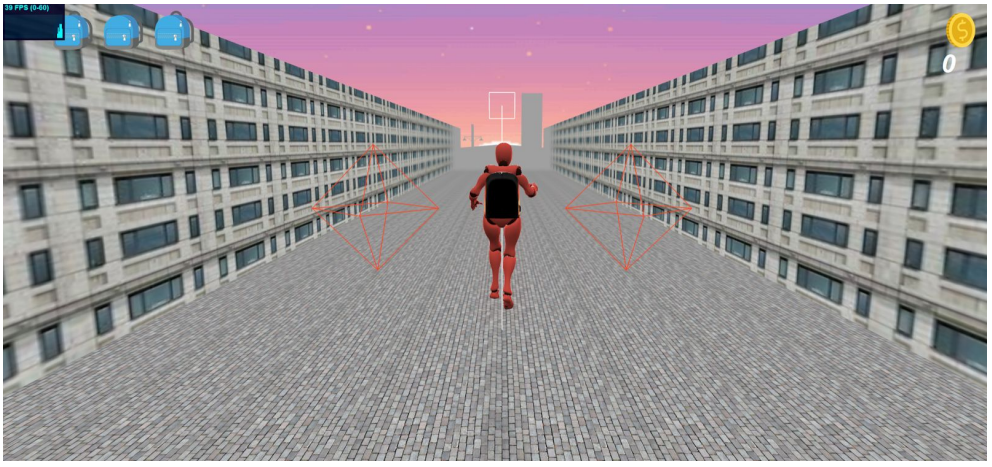
For all the obstacles that are rendered from a *GLTF Model*, the animation is triggered after a collision with the player happens: the same reasoning as before has been applied to defining all the collision animations, by using the hierarchical structure of the model involved and by applying *Catmull-Rom interpolations* and/or *easing functions*. In particular, for *the car model*, a polynomial *Quartic function* has been used to animate *the chassis*, while the wheels are eased in with a *Bounce effect* (after the previous animation has been completed) and eased out with a *Cubic function*.



Also for the tree and the lamp model, a *Bounce function* has been used to ease out the animation, while for the gazelle animation an *Elastic function* is repeated infinitely to make the model oscillating while moving its legs.

5. Lights

Now we will cover up all the different lights involved in the scene. By enabling the debug mode, it's possible to see all the helpers attached to the different lights, in order to see all their relevant features.



City scenario at sunset, with light helpers enabled.

5.1 Hemisphere Light

The *Hemisphere Light* is defined by two parameters: the sky colour and the ground colour. If an object surface is pointing up, it will be multiplied by the sky colour; otherwise, if the surface of the object is pointing down, it will be multiplied by the ground colour.

5.2 Directional Light

The *Hemisphere Light* gives to each colour the influence of the sky and the ground. However, we still need a light that represents the sun on the scene. The *Directional Light* has been used for this purpose, and it's available in two different formats to represent two different configurations of the scenario: the first, which represents the *morning*, is set just behind the character; while the second, which represents the *sunset*, is set at a higher distance from the player.

5.3 Point Light

To give more shininess to the models, in particular the coins, two *Point Lights* have been positioned on the two sides of the scenario: the light is emitted from both points in all directions, with smaller intensity.

6. Conclusions

This project has been a useful experience to see all the topics during the course being integrated within a web videogame, developed using some of the many libraries that are built on top of *WebGL*.

6.1 Performances

Since the scene renders many objects at the same time, the *Physijs library* may not work properly with *slow connections* or *web browsers* with a limited amount of resources: to optimize the application, each model is removed and from the scene as soon as it goes behind the player and, whenever possible, the object calls the *dispose method* to remove all its resources from the buffers and the shader programs.