SAPIENZA
UNIVERSITÀ DI ROMA

JAKILADA

Interactive Graphics

Corso di Laurea in Engineering in Computer Science

Candidate

Antonio Marigliano - Daniele Michelori - Ilaria Ponzianelli - Gianmarco Salvi
ID number 1745765 - 1757243 - 1745835 - 1763156

Academic Year 2020/2021

JAKILADA

Bachelor's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: marigliano.1745765@studenti.uniroma1.it - michelori.1757243@studenti.uniroma1.it - ponzianelli.1745835@studenti.uniroma1.it - salvi.1763156@studenti.uniroma1.it

Contents

Chapter 1

Introduction

As final project for Interactive Graphic course we have decided to create a web-based game using some of the most common libraries in web graphic field: Three.js and TweenMax (an extension of Tween). The aim of our game is to take control of an airplane and try to reach the farthest possible distance in a selected scenario breaking the previous record, avoiding animals or spaceships during the fly. The airplane can be moved up, down, left and right to dodge enemies using keys from the keyboard. In this report we will discuss about the technologies and libraries used, the models created for the game and the most important functionalities developed.

Chapter 2

Technology

We used different languages, libraries and tools to develop this project.

2.1 Languages

The backbone of our project lies on the three most known and spread languages web development field:

- **HTML**: *HyperText Markup Language* is the keystone of every web application. We have employed *HTML5*, the latest version of the language which was developed according to W3C standards, mainly for handling the containers for the canvas context and the various menu. In addition, the features provided by HTML5 allow us to easily implement the audio for the background music and other noises produced in game (like the ones made by obstacles during collision).
- **JavaScript**: it is a programming language commonly used for handling objects, events and general dynamic behaviours. This has been the most used language in our project, (every third party library is based on this language).
- **CSS**: *Cascading Style Sheets* is a style-sheet language utilize for the definition of the presentation of the document, strictly combined with HTML5.

2.2 Libraries

We have chosen **Three.js** and **TweenMax.js** as our main libraries for this web game, following the indications and requirements for the development of the project. Regarding their features and reason for choosing them, we will discuss briefly below.

- **THREE.JS**: It is a cross-browser JavaScript library and an Application Programming Interface (*API*) widely utilized for 3D Graphic Modelling in Web Browser using WebGL. We have decided to use this library as the backbone for our graphic model, instead of others like *Babylon.js*, due to the following reasons:
  - Is older, so it includes more features and is better documented than other libraries;
  - Offers a canvas renderer as a fallback when WebGL is not available;

  - Supports physically based rendering (*PBR*), allowing for real life quality material and lighting;
  - Has a wide community of supporters and case studies as examples;
  - Has a lot of third party integrations and support for every browser.

  Regarding the third party integration, we have employed **threex.keyboardstate**, which is a threex game extension for three.js which makes it easy to keep the current state of the keyboard.

- **TWEENMAX.JS**: *TweenMax.js* is an extension of TweenLite, which adds to it many useful functions and features like *repeat()* and make it easy to load third party plug-ins. It offers the same base features and syntax of *TweenLite*, with the addition of more optimization for speed and simpler design for usage. Its "technological" core, as goes for TweenLite, is the Tween.js library. This one is a JavaScript tweening engine design for handling animation smoothly in canvas contex. We have chosen this library because:
  - Its simple and effective design, allowing us to employ functions easly across the thousands lines of code;
  - Its possibility to integrate it with other plug-ins;
  - Its simple and readable documentation, with many case studies brought by the community.

2.3 Other Useful Tools

In order to develop, deploy and test our web game we used two useful tools:

- **GitHub**: It is a provider of *internet hosting* for software development and version debugging using *Git*. It provides the possibility to create multiple branches for parallel working on single project and multiple operations in order to handle repositories and branches (like push, pull and commit);
- **GitHub Pages**: It is a static web hosting service provided by GitHub since 2008. It provides developers the capability of building fast and simple websites, with full documentation available on the corresponding repository.
- **Visual Studio Code**: We decided to us it as editor for the development of our project because it allows to underline the syntax of every programming language, provides debugging support and allows to interact with Git.

Chapter 3

Game Design

In this chapter we are going to analyze the concept design of our game and its main mechanics. In particular, we are going to focus on a user perspective.

3.1 *Fly, as far as you can!*

The base idea of this web game is to take the control of an airplane on three different scenario and reach the farthest possible distance, while avoiding obstacles like animals or spaceships. The users may use both the classical *W-A-S-D* control set to move the airplane, or could press the arrows on the keyboard. The movement of the airplane on the x-axis triggers a little animation that incline the airplane to a side or another while key-arrow is pressed. In addition, this movement leads to a slightly faster or slower rotation of the entire world, which of course influence also the speed of the obstacles. Users start the game with three *lives* and if they collide an obstacle, then they lose one until no one left and the **game-over** display will be shown. Users can recovery lives thanks to a *bonus life* that appears every 500 meters travelled.
As a side note, a collision will trigger an *invincible status* that allows users to continue their fly for a few seconds while being totally immune to other collision.

3.2 Options and Menu

The first page that will appear to user is the main menu, with the typical buttons like *New Game*, *How To Play*, *Select Scenario* and *Credits* while a small buttons for switching the audio on/off is placed on the bottom-left of the screen. As it is instantly visible, the selected scenario, which is by default the *Desert* one, will appear on the background while rotation on itself, but without obstacles and the airplane. This could easily show the chosen scenario to users and then let them decide which one to use according to their wishes.

3.3 Scenarios and Animals

There are three distinct scenario: *Desert*, *Countryside* and *Moon*. We have modelled them upon a similar structure, as it is easily understandable while playing them, but we will discuss them in the next chapter. Lastly, there are two kind of animal for the first two scenario, which are *Condors* and *Ducks*, while the scenario in the space is populated by *Spaceships*.

3.4 Enemies and bonus spawn logic

The enemies spawn is managed in the *spawnAnimals* function. In this function a predefined number of enemies (*nAnimals*) objects is created and pushed into *animalsArray*, depending on the current scenario.
Enemies are inserted outside the visible scene, with the following logic to distribute them in a random but uniform way during the game: the X axis is divided into steps (the green ones in the image, each one "step Lenght" long) and each step is split along Y axis in "nSpawnPerStep" slots, which is a variable that defines the number of animals to be spawned in each step, and grows with levels. In this way enemies will increase in number and also in their speed along with levels. Each enemy is positioned inside a slot, and is assigned with coordinates that are chosen randomly between the X and Y axes boundaries of that slot.

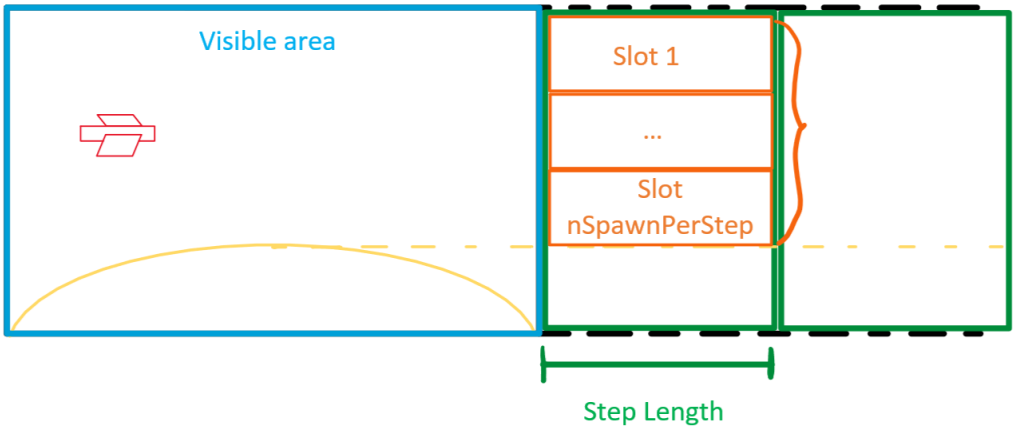

Figure 3.1. Spawn scheme

The enemies appear from the right side and cross the screen to reach the opposite end. When they exit the visible area they are removed from the scene and re-collocated in the farthest position along X axis, way out from the visible scene. This logic allows to build just a few enemies and reuse them inside the scene, saving memory e improving performances.

The bonus element has the same logic of the enemies, but it appears every 500 score points and bounces up-and-down through the length of the screen.

3.5 Audio

The audio is enabled and disabled from the icon at low left angle. Every sound (the collision with animals, spaceships and bonus lives, the backtrack of the game and the game over sound) are managed in html file.

Chapter 4

Models and Light

Firstly, we want to show the main models for our web-application and also a brief overview of secondary models used to populate the scenarios.
The majority of the functions was imported by Three.js, while colors and materials were sometimes gathered into a single initial vector section for a modular approach in programming.

4.1 Airplane

The most important element of this game is the **Airplane**, which is the only model controlled by the user and used to actually "play the game". It is constructed using a *hierarchical model* at high level through the function *THREE.Object3D()* provided by Three.js. Multiple *BoxGeometry* and chained one to another alongside their respective *MeshPhongMaterial* in order to create the final result visible during the fly. All of its sub-part are purely squared, recalling "old school" games, with low number of polygon on screen. As it can be seen into the code, we have modified the position of some vertices and the proportion of the plane for our purposes. The model has been built from the ground during development, without importing another one already done. Lastly, we decided to use a simple concept for the airplane according to the minimal and simple *"look"* we have built around this game.

4.2 Condors and Ducks

Because of their similarities, we are going to analyze both of them into a single section. They are built with multiple geometry: *BoxGeometry*, *CylinderGeometry* and *ExtrudeGeometry*; while the choice of the material lied again on *MeshPhongMaterial*. The mesh originated by their combination where then scaled and manipulated in order to obtain a stylized version of a **Condor** and a **Duck**. To make animals adapt for the scene, we rescale the components and we reposition them.

4.3 Spaceship

The **Spaceship**, or simple *Ship*, is the last antagonist of this web-game and populate its third scenario. It is created with a combination of *LatheGeometry*, *SphereGeometry* and *TorusGeometry*, while the choice of the material this time lied upon *MeshBasicMaterials* because of its nature that perfectly match our goal for modelling the spaceship. As for all the other models, geometry has been resized in

order to obtain the desired structure of the ship, especially regarding the spiral form on top.

4.4 Desert, Countryside and Moon

In this section we will analyze all the scenarios we have built in the game. We will talk more generally about base structure, since it is shared among the three landscapes, and then about the elements composign the background.

4.4.1 The Cylinder Structure

The base geometry upon which are built our scenarios is an enormous *Cylinder* upon which is placed the *Camera* and the other models. The choice for this structure has been driven by the game design itself, because we wanted to create an "infinity" loop world populated by animals or ship which could represent also the idea of a minimal world. So, we choose this geometry for its simplicity in game development, in fact the *loop* is created by the rotation of the cylinder and all the object around it.

4.4.2 Landscape Models

What truly distinguishes one landscape to another is the presence of the elements that populate it, like the *cactus*, the *trees* or the *rocks*. A brief explanation of every secondary model will follow below:

- **CACTUS**: the *Cactus* class combines *ExtrudeGeometry* and *MeshPhongMaterial* in order to create the structure of a real-world cactus, but on a minimal and stylized perspective;
- **TREE**: this class recreate a minimal tree by chaining *ExtrudeGeometry* and *SphereGeometry* with *MeshPhongMaterial*. This model has been duplicated in **Tree2** in order to elaborate a different type of tree to give the "world" a bit of randomness;
- **CLOUD**: this object replicate a minimal and stylized cloud, thanks to the use of simple *SphereGeometry* combined with *MeshPhongMaterial*. In addiction, for a better looking perspective, we have implemented a random function for creating the clouds, while putting some of them on the background and other closer to the screen. Moreover for the countryside scenario we used more clouds than in the desert one;
- **BUSHES**: they have been designed in order to populate the *countryside* scenario, they are random and separated from trees. A bush is created as a combination of *SphereGeometry* with *MeshPhongMaterial*, using different scales;
- **STAR**: it is a simple and effective class used for populating the "sky" of the *Moon* scenario, as it can be seen by running the game. It exploits *BoxGeometry* and *MeshPhongMaterial* with some mathematical manipulation for creating the "slide effect";
- **ROCKS**: the *Rock* class employs a singular *BoxGeometry* combined with *MeshPhongMaterial* in order to create the rock that populate the moon.

They are created using a random function and positioned on the floor of the cylindrical structure to form lunar mountains;

- **HOLE**: this class is used in order to replicate the craters which caracterize the moon. This is done by combining *CircleGeometry* and *MeshPhongMaterial* for the creation of a stylized crater.

4.4.3 Sky Holders

For the creation of the sky we follow a modular and hierarchical approach, by creating *THREE.Object3D* that will hold animals, clouds and stars depending on the scenario:

- **DESERTSKY**: as the name may suggest, is the class used for holding the clouds and condors in the first scenario. It is created as a simple container with a background color that resembles the desert sandstorms.
- **COUNTRYSKY**: the *CountrySky* is an empty container with lightblue background color, designed with the purpose of holding both the clouds and the ducks that populate the scenario;
- **SPACESKY**: its the container created for the moon scenario and holds the multiple star object that defines the space on the moon.

4.5 BonusLife

This class is responsible for the creation of bonus lives that will be positioned into the scenario every 500 meters. It works by combining *BoxGeometry* with *MeshPhongMaterial* in order to replicate a simple 3D cross in green material. This class also implements itself some functions responsible for the animation of the object, as it will be explained in the next chapter.

4.6 Light

The lights implemented into a scene, play always an important role into creation of suggestive and impressive environment. In our web game, we have used the *THREE.HemisphereLight*, *THREE.DirectionalLight* and *THREE.AmbientLight* functions in order to create an "in-game" sun and a "shadow" effect for every object present on the screen, from the airplane to the flying obstacles. We have decided to position the "sun light", which is represented by the directional light that acts like a sort of real world sun, on the top of the screen, while defining the area of the shadow projected by animals and ship.

4.7 Textures

We have added textures only on the airplane to not make the project heavy. We have decided to insert a steel texture to make it shine in a metallic way. In particular, we used a red texture for the body and a white texture for the engine. We also added a glass texture for the cabin of the pilot to give transparency to this component.

Chapter 5

Animations

In this chapter we will illustrate all the animation present into the game, from "active one", which is the movement of the plane, to the "passive ones", that defines the rotation of the scenario and the movement of its objects.

5.1 The Airplane

The player can move the airplane into the scene by using the classical layouts: *W-A-S-D* or the keyboard arrows. When pressed, they trigger the multiple function that handle the airplane "movements".
Regarding the "loop control", the *updateFCTS* array is called at any change of frame into the *requestAnimationFrame* JavaScript function that allows to execute code on the next available screen repaint. It is called at every frame in order to check if a key is pressed. When this occur, the *airplane.mesh.position* is changed, on both axis, following a mathematical formula which can be easly found in the code.
The movement on the x-axis is combined with the use of **TweenMax** library in order to get a smooth animation of the plane rolling on both side. It is stopped and resumed when the user pause and unpause the game.

5.2 Condor, Duck and Ship

The flying "enemies", or obstacles, follow specific functions that handle their movements into the scene. In particular, it has been implemented a single function, called **moveWing**, in order to handle the position of an object with respect to the current chosen scenario. As it can be easily understood from the code, the movement is implemented by changing the rotation and the position of the object mesh. The position of them is handled by the **moveAnimals** function, constantly called into the loop function.
In moon scenario we have a function that allows the rotation of the entire object, the spaceship.

5.3 Life

As we mentioned before, every 500 meters a bonus life appears from the right of the screen and, if taken, gives the user the possibility to regain a life. It moves according to a prototype function of the class *BonusLife* which is called **handleOn-**

Scene. This function let the mesh of the bonus life float up and down into the scene by simply changing its *mesh.position.y*

5.4 The Scenario and its Background

As we have mentioned in other sections, both the scenario itself and the objects that populate it are constantly moving. Their animation is simple, they are mainly handled into the loop function through basic assignation or modular function. The scenario itself is animated by **backgroundMovement** function, which would increase the *currentscenario.mesh.rotation* by values multiplied to *airplaneXpos*, *game.level* and the addition of a fixed value. Meanwhile, the container for the flying object, like desertsky, is costantly rotating by the same parameters that multiply the currentscenario mesh but for different fixed values.

5.5 Collisions

The animation concerning the collision among airplane and enemies is handled by the class **AnimalParticles** and **LifeParticles**, in particular by their prototype functions **explose**. This functions work by taking every particles genereted by the class itself and then sending them into different random direction into the space, as it can be seen into the code. In this case, we have used *TweenMax* library in order to create a smooth animation for the particles propagation, which is also paused and unpaused during the game pausa/unpause. We have decided to reuse the same class and function for the flying obstacles and lives for a more modular approach changing only colors.

5.6 Invincibility

Linked to the collision itself, it is a small "animation" that combines timers and *airplane.mesh.visibility* in order to return the effect of a flashy plane while the invincibility mode is active for 3 seconds after . During this time, the player can have multiple collision without risking any additional lives.

Bibliography

[1] https://threejs.org/

[2] https://github.com/jeromeetienne/threex.keyboardstate/

[3] documentation for TweenMax library: https://greensock.com/docs/v2/TweenMax