SAPIENZA
UNIVERSITÀ DI ROMA

Master of Science in Engineering in Computer Science

Interactive Graphics

Final Project: Monkey Go!

Presented by:
Andrea Fantoli 1467336

Submitted to:
Marco Schaerf

Academic year: 2017/2018

Index

Introduction

For my project I decided to develop a simple game, of the type "endless runner", using all the notions such as: camera, light management, textures, hierarchical components for the models, learned during the course and the *THREE.js* library, where the protagonist is a small monkey that is running on a road that runs endlessly through a jungle. The name I chose for my game is precisely "**Go Monkey**!", where our monkey will deal with 4 types of objects during his run: 2 malus that are stones and bombs and 2 bonuses that are bananas and hearts. The aim of the game is to survive as long as possible by dodging, moving to the left or right or by jumping, the rocks, which will make us lose a life, and the bombs, which instead will take two, trying instead to collect the bananas that will give us an extra life every 10 and the hearts that will give us directly one more life. The game ends when the lives will reach 0, there are no time limits, the better you become and the more you will get far, the number of meters covered is marked in real time at the top of the screen and at the end of the game will identify our score. At the end of the game will appear the "*Restart*" button to start a new race and support, during the game, pressing the *'space bar'* we can pause the game or if we are not satisfied directly restart it by pressing *'R'*.

Project

File components

For my game I decided to have everything on a single file ***game.html*** where I change the style of the various elements through the file ***game.css***, all the part related to the game itself is located within the file ***game.js***. Starting the *.html* file we open the game itself where after a few seconds of waiting due to loading the models, clicking with the mouse we can interact with the monkey, all movement occurs thanks to the arrow keys that are on the keyboard and to recover the mouse pointer we must press *'esc'*. Below I will describe the content of the listed files dwelling more precisely on the *.js* file that contains all the details of the game.

Game.html

In the html file I declare all the objects I'm going to use in the game:

1. The **divs** *'container'*, *'blocker'* and *'instructions'* are used to manage the loading screen of the models that appears before starting to play.

2. The various **labels** represent what we will see in the game screen: *'command1'* and *'command2'* make visible in the upper left corner the game commands illustrating all the keys that you can use, *'command3'* is the step counter updated in real time to show the progress of the game that will be located in the upper middle of the screen, *'command4'* and *'command5'* are the counters of lives and bananas that are located in the upper right corner above the images of the heart and banana that I will load in the file *.js*, *'command6'* represents the word "*Game Over*" that will appear at the

end of the game and finally *'command7'* will do the same for the word "*Pause*" when I decide to pause the game.

3. The *'restartButton'* **button** will allow us to start the game again, it will appear only and only when the game is finished together with the word "*Game Over*".

4. As for the **scripts** used I will use the library *'three.js'* downloaded and present in the libs folder, the online library present at the address specified for *'three.min.js'* and finally I will call my file *'game.js'* for everything related to the game.

Game.css

The *.css* file collects all the styles used for various components, specifying the positions on the screen, the size, colour and type of font used for the writings and everything related to the modelling of these objects. As for the font used for the writings I downloaded and incorporated a font not present in the standard collection, choosing one more suitable for a game.

Game.js

In more detail, I'll try to explain everything I implemented at the code level to build my game, focusing on the code for the individual components of the game:

1. **getPointerLock()**: the first function that I managed is the one concerning the loading screen, through this specific function that once clicked on the screen to start playing, the keyboard will have the exclusive as it will be disabled the mouse pointer, so now we can devote ourselves exclusively to the game; when we want to reuse the mouse we will just press the *'esc'* key and the keyboard will lose its exclusivity.

2. **createScene()**: to switch from a 2D screen to a 3D one I start to declare and set everything that will allow me to enjoy this experience. I create the scene, specify the game room, initialize a sun as the primary source of light and set the shadows for that source, as support functions imposed the playability via keyboard (*'document.onkeydown = handleKeyDown;'*) and create a music file (*'songTheme = new sound ("song/theme.mp3");'*) that I will manage later to have a background music during the game making it more enjoyable and captivating the game mode. In the creation of the specific scene the texture that will cover the plane used as a street, using an image found on the internet, and the plane itself by changing position and size so that it can best represent a long way in the jungle, after adding it to the scene I create a plan copy of the previous one, this I do because then in the management of the flow of the road to give the feeling of an "infinite" plane will take the place of each other. Finally, I initialize the render with the settings specified above for the size of the screen and shading also activating the function that allows me to use the objects created in the .html file and set in the *.css*.

3. **sound(source)**, **loadSound(name, time)**: the following functions, are two standard functions regarding the loading of an audio file, take advantage of the source path of the music file and the name associated with the object I created earlier (*songTheme = new sound ("song/theme.mp3");*).

4. **addIcon()**: in this function I create two circles using the library *THREE.js* and then put on the textures of a heart and a banana (taken from two images *.png*) giving the idea of having loaded an object heart and an object banana. These two icons will serve me thanks to their respective labels to identify the progress of the game about the remaining lives and bananas collected by the monkey.

5. **addPlane()**: as already explained when creating the plane, this function actually scrolls: when the position of the plane reaches a certain value of z (it approaches the screen) it is always moved back on the z axis; the clone plane of the original will be useful from the moment when, by force of scrolling, the plane could disappear and having a copy of it at our disposal, this will not happen. The scrolling speed is given by the variable *'obstaclesMove'* which is also the variable that determines the speed of objects that will be fundamental in the function that increases the difficulty of play.

6. **addModel(), addMonkey(), addRock(), addBomb(), addBanana(), addHeart()**: these are all the functions that deal with adding the various models to the scene, since except for the main one of the monkey are almost all the same I will limit myself to explain in detail only this. I start by saying that the models I used are *.json* models that I found on the web and that I modified according to my needs thanks to the editor site "*https://threejs.org/editor/*", the models I used are 5: monkey, rock, bomb, banana and heart. As for loading, I used the *load()* function provided by the *THREE.js* library where I create the object variable related to the specified model, I manage the position, rotation and size within the scene. As for the monkey I chose a model composed of several parts, specifically: right and left hand, left and right foot, left and right ear, tail, head and torso, to exploit during the movement the hierarchical model as required by the design specifications and as learned during the second homework. After having associated each component to a variable, I manually move the tail otherwise during the movement I imposed it would have given the impression of detaching from the body. The second and third parameters of the standard function represent the loading percentage of the model and its success or error.

7. **handleKeyDown(keyEvent)**: after having enabled the use of the headboard in this function I associate a certain activity to the keys I have indicated: by pressing the right and left arrows I will allow the monkey to move horizontally on the plane, within certain limits set in the declaration of variables, by a certain amount equal to the value of the variable *'monkeyMove'*, during the movement I also rotate both feet and I move up and down the tail of a quantity I set, managing the reposition()

function through a *'setTimeout()'* to bring everything back to the starting position; pressing the up arrow, beyond the movements of feet and tail, I allow the monkey to jump, in this function I change the position on the y-axis of the monkey within a certain range of jump, during the flight phase I make null the possibility of a second jump through the variable *'jumping'* and through *'left'* and *'right'* I will not allow the monkey to move horizontally and also guarantee the invulnerability to the passage over obstacles, with a timeout and the *downJump()* function I manage the return to the ground and also with *stopJump()* prevented a second jump in the immediate to not be able to avoid the obstacles always with this movement; by pressing the space bar I pause the game, if the variable *'stop'* is false, stopping the animation, the step count, the music and making the label "*Pause*" appear in the middle of the screen, resuming the bar reactivates everything reversing the process described above, and finally pressing the *'R'* restart the game with the command '*window.location.reload();*' which reloads the current page, i.e. the page of the game.

8. **downJump(), stopJump(), reposition()**: these are all the support functions I have previously mentioned: *downJump()* brings the monkey back to the ground after the jump, calls *reposition()* to rearrange the rotated model components, rehabilitates the lateral displacement and makes the monkey vulnerable again; *stopJump()* rehabilitates the jump by setting *'jumping'* to true; *reposition()* finally simply takes care of bringing the feet and tail back to the starting position cancelling the rotation due to the movement.

9. **chooseModelToLoad()**: now let's move on to loading the models and managing them. In this first function genre a random number between 1 and 100 to manage the distribution of loading of the various models on the plane: a number between 1 and 65 corresponds to the rock, between 66 and 80 to the bomb, between 81 and 98 to the banana and finally between 99 and 100 for the heart. Through the 'code' function I was able to associate each model with a code that I will then use in subsequent functions to manage the collision with a specific object.

10. **setModel(obj), loadModel()**: these are two functions that work together: *setModel(obj)* is the function that deals with randomly positioning in a certain range the models on the rotating plane with respect to both the x-axis and the z-axis, this function at the time of loading the model when it is called places it in the positions generated by actually adding it to the scene; *loadModel()* thanks to the previously saved code associated with the object, allows me to differentiate the model to load by calling precisely the previous function with its code compared to the generated object, the method closes with a for that I spin as many times as there are objects I want to create (30), inside it in addition to loading models thanks to *setModel(obj)* save them all over a vector that I will use to propose objects on the plane.

11. **obstacleManagement(), impact(code)**: after creating and arranging the models on the plane we move to the phase of collision management with these two methods. The first *obstacleManagement()* takes one by one all the elements loaded on the vector, in the first if it manages the repositioning on the z axis of the element, before it disappears it brings it back to the bottom regenerating randomly its position on x, moreover it leverages it from the vector putting it back in tail; done this if the monkey was actually loaded, on *'distanceZ'* and *'distanceX'* it calculates the minimum distance that there must be as long as a collision occurs managing if, if the distance is satisfactory the object is removed from sight (*'obstacles.position.x = 2000;'*) making it disappear from the screen and calls up *impact(code)* to manage the effects of the colliding; to manage the collision of the model of the bomb I had to use a dedicated if because of the specifications of creation of the model itself that differed with the others, making tests I saved the actual positions of collision making sure that if *'distanceX'* was equal to those values then the collision would have occurred. *Impact(code)* is a simple method that uses a switch where according to the model's code it manages the effect of the collision between monkey and objects: if it hits a rock it loses a life, if it takes a bomb it loses two, if it takes a banana it updates the banana count and every time it takes 10 it increases the life count by one, finally if it takes a heart it gains an extra life.

12. **gameOver()**: as the name of the function itself says, this represents the call at the end of the game, when the monkey loses all its life this function is called that blocks the animation, stops the music and the step count making the label appear with the words "*Game Over*" and the restart button that contains the code to reload the current page and start a new game.

13. **countPasses(), increaseMove()**: the first function acts as a step counter using the logic of a chronometer associates every second a step and displays a meter in the pedometer on the screen in real time, calls through a timeout every second to update the variable that acts as a counter, relies on the two functions *startCounter()* and *stopCounter()* to operate and stop the counter. *IncreaseMove()* instead allows me to increase the speed of the plane and obstacles scrolling every time 50 steps are taken (this if is in the render).

14. **render()**: last is the function of the *render()* where the animation is called and all those functions that must be repeated once it is called. Inside I manage the *startCounter()* and the loading of models on the scene only if all have been initialized; the increase in speed with *increaseMove()* every time you make 50 steps; the call to the function *gameOver()* setting to true its variable if the lives fall below 0; adds the plane with *addPlane();* activates the collision management with *obstacleManagement()*; calls to itself the scene and the camera; and as a last operation if the variable *'game_over'* is false makes active the animation that will be interrupted only during pause or game over.

References

- ✓ http://davidscottlyons.com/threejs-intro/#slide-0
- ✓ https://threejs.org/
- ✓ http://blog.cjgammon.com/threejs-models
- ✓ https://gamedevelopment.tutsplus.com/tutorials/creating-a-simple-3d-endless-runner-game-using-three-js--cms-29157
- ✓ http://texturelib.com/
- ✓ https://clara.io/
- ✓ https://docs.microsoft.com/it-it/windows/uwp/get-started/get-started-tutorial-game-js3d
- ✓ https://threejs.org/editor/
- ✓ https://free3d.com/
- ✓ https://www.turbosquid.com/
- ✓ https://www.w3schools.com/css//