SAPIENZA
UNIVERSITÀ DI ROMA

Interactive Graphics

Final Project

THE HANDLEY RUN

Professor:

Marco Schaerf

Student:

Federico Boarelli (ID 1549662)

Summary:

Introduction:

The Handley Run is WebGL based endless videogame, where the player impersonate the pilot of an Handley plane (a first world war English bomber). The scope of the game is simple: let the Handley arrives as far as possible. During this infinite over sea tour, the player must take care of the amount of fuel available on his plane: if it will finish, then there will be the game over. The game's world is populated by various objects, both active and passive: for the first one, we have the red *bad* spheres, which will increase the loss of fuel of the Handley, and the green *good* spheres, which augment the amount of fuel available, while for the second kind of object we have the clouds, that are randomly generated all around the world and, of course, the sea. The aim of this project, created with the Three.JS and TweenMax libraries is to show that even without the need of models from external source and very complex logic, is possible to realize a basic game without sacrifices quality and entertainment and, more important, it is possible to let it be available from any web browser. The game is runs in a circular way.

NB: the game perform good on Firefox and Microsoft Edge, but for some compatibility problem it does not work in the correct game on Google Chrome.

The project

Structure:

The files and the directories contained in the main folder are structured in the following way:

- The *index.html* file that is the file where all the components of the project get together. This can be view as the container of the entire project, where all the libraries and the static resources are loaded at runtime
- The *js* folder is the hearth of the game where all the logic, written in JavaScript, is contained: the *game.js* contain all the logic developed to create and animate the game's world following the classical pattern of a WebGL application. The other two files, *three.min.js* and *TweenMax.js* are the libraries used to realize this project: the first one is used to create the scene, the lights, the models and the loop of the game, meanwhile the second one is used to give a more realistic physic approach to the game
- The *css* folder contain two css files where we define every aspect regarding the component style, like fonts and UI elements.
- The *res* folder is where we put all the static files, textures and soundtrack, that we use to make the game more enjoyable

User interaction:

The user has the control of the plane during the whole game: the interaction of the plane is implemented throw a listener over the mouse that let the user to execute two actions. The first one is, clearly, the possibility to change the position of the plain on the y axe, so going up and down just moving the mouse (or the finger on the mouse pad). This implementation was done to make easy the development of the touch commands to bring the experience on the smart devices. The second action is executed over the x axe and let the user to expand or decrease the size of the camera's view: this allow the player to have a clear vision of the whole scenario and for a performance reason. In fact, having many objects on the screen make the bigger view size more expensive in term of resources, meanwhile the small improve the performances. The movement are related to the plane as if it is a unique object, meanwhile it is a hierarchical model where many components, like the propeller and the player's hair have their own complex animation, as the other objects of the game (the sea, the clouds and the spheres). Other interactions happened when the player catch the green *good* sphere, that recharge the plane's fuel, and when the player is hit by the red *bad* spheres: in this case, the plane is pushed a little bit behind over the x axe with a custom animation. All the animations relative to the plane were realized with the TweenMax library, which allow to get extreme realistic physic of the plane. The last interaction that the player do is the one when clicking on the *Replay* message after the game's end.

Code:

Let's now see the logic of the functions that are part of the game. In general, we define a set of variables to handle all the object of the game keeping them as separated as possible, to avoid unclear code or duplicate. Then we construct with dedicated functions the environment that will represent the final product.

<u>**Main functions:**</u>

resetGame():

This function is used to restore the value of the all variables of the game to the initial one. It is called after that the play click on the *replay* button to start a new match. We assign to variable contained in the object *game* all the initial values.

createScene():

In this function, we setup up the scene for handle the game's world: we create a new ThreeJS *scene* object defining the *camera* and the *renderer*. All the previous variable mentioned are global variable.

createLights():

we define three light objects: *hemisphereLight*, *ambientLight* and *shadowLight*. Since our world is a circular one, the HemisphereLight implemented in ThreeJS is the most natural way to obtain, let's say, a global circular illumination. The *ambientLight* define instead the global illumination, with particular reference to the lamination, meanwhile the *shadowLight* is used to obtain the shadow for the object over the sea.

<u>Hierarchical Models:</u>

In this game, there are two hierarchical models implemented: the *pilot* and the *airplane*. The first one, is composed by one father mesh that is called *pilot* followed by the other components: *body, face, hairs* (which can be consider as another hierarchical model due to the fact that as many components inside, in particular the *HairSideL* and the *HairSideR* which are essential for the hairs animation) *glass* and *ears*. The Airplane model is composed by the following objects: the main mesh called *airplane,* then by *cabin* (which is a BoxGeometry object altered to get a more realistic aspect) *engine, tale,* two *wings* and four *supportWing, pilotShield* , *propeller* and three *wheel*.

<u>Other models</u>:

We define all the other entities as follow: we create a function to define the object and then, throw the *Object.prototype* inheritance we assign to any object its specific animations and properties. So let's now discuss briefly the objects and their properties.

Sea() and Sea.prototype.moveWaves():

The sea is realized as a cylinder: this choice was done to implement in the most suitable way the circular game progression. We initialize the Sea and the we implements with the inheritance the *moveMoves* (that performs the movement of the waves) to make our Sea looks as most as similar to the real sea.

Sky(), Sky.prototype.moveClouds(), Cloud and Cloud.prototype.rotate():

We define the Sky and we insert in it a fixed number (20) of Clouds: every Cloud is an object and implements its own random animation throw the Cloud.prototype.rotate() function, and this function is called for every clouds in the Sky.prototype.moveClouds() that, like for Sea.prototype.moveWaves, its essentially a loop to keep the animation of the clouds active. Every Cloud is composed by a random number of cubes and the animation is applied to the resulting mesh: we can consider then the Cloud as a simple hierarchical model.

Enemies and Coins of fuel:

The enemies and the coins share, in a certain way, the same structure: we define the objects like the previous one, then we implement three functions: the enemyHolder() and the coinsHolder() that are use to keep trace of the active entities, the EnemyHolder.prototype.spawnenemies() and the CoinHolder.prototype.spawncoins() where we create and add the objects to the active entities and the EnemyHolder.prototype.rotateenemies() and the CoinHolder.prototype.rotatecoins() where we handle the collision with the player and the animations. For the collision, we define another entity called Particle (which has its own prototype functions then) to use the TweenMax libraries to create a more realistic explosion of the spheres. In case of collision with the coins, the player just gains energy and the coin explode, while the player crash with a enemy sphere, the sphere will explode and the player will do a sort of *back jump*: this properties are handled in the .rotate() functions.

loop():

Here is where we let the game start: we check the game status (*playing* or *waitingReplay)* and the we perform the action. We take care here of the spawn, the speed, the animations and the level of the game. If the game is over, a special animation is applied to the plane. In this status, the world will continue to move, meanwhile the camera will be freeze on the point where the player lose. We call here the WebGL requestAnimationFrame().

init():

Here we setup all the UI for the start and we create all the objects and the scene. The function end with the call of the loop() function.

<u>Other functions:</u>

handleWindowResize():

This function is used to handle the resize of the browser window

handleMouseMove(event) and handleTouchMove(event):

Those functions are used change the position of the *mousePos* (the mouse pointer) in order to change the position of the plane following the input of the player

create*Objcet*():

We have designed one of this function for any object present in the game: them are all called in the init() function to create the game's world

updateDistance():

This function is used to update the distance that the player has completed in real time. The distance's value will *freeze* in case of game over

updatefuel():

This function is used to update the amount of the fuel of the plane in real time: the fuel will decrease during the game and after collision while it augment if a green sphere is taken by the player. The game over will occour only if the player finish the fuel

updatePlane():

This function is used to update the position of the plane in real time and that the animations of the pilot and of the plane are constantly executed.

normalize(v,vmin,vmax,tmin, tmax):

This function is used to normalize the speed and the position of the plane with respect of the size of the window, in order to maintain the position like if it is always the same

CSS:

We have two CSS files: *game.css* and *world.css*. While the second is defined just to give a global palette color to the game, in the first one we define the style of all the UI components in the game. We define a *mediascreen* property too, to handle the responsive scenario.

Conclusions:

Building from scratch a game is not *mission impossible*: thanks to powerful libraries like Three.JS and TweenMax, and a little bit of fantasy, is possible to create any kind of project that you may think about. The scope of this project was to have fun and increase the knowledge of the WebGL technology and to show that is possible to create and animate complex hierarchical model without using external models inserting them in a complex world composed by lights, physic, textures and with a simple but not so trivial logic behind the scene that coordinates all the things. This project was realized for the Interactive Graphics class held by professor Marco Schaerf, academic year A.A. 2017/2018.

References:

- https://threejs.org/docs/
- https://threejs.org/editor/
- http://texturelib.com/
- https://github.com/mrdoob/
- https://tympanus.net/codrops/