Le Petit Prince

Mele Chiara - ID 1259767
Palleschi Alessia - ID 1535167

a. y. 2017-2018

Si tu viens, par exemple, à quatre heures de l'après-midi, dès trois heures je commencerai d'être heureux. Plus l'heure avancera, plus je me sentirai heureux. À quatre heures, déjà, je m'agiterai et m'inquiéterai; je découvrirai le prix du bonheur! Mais si tu viens n'importe quand, je ne saurai jamais à quelle heure m'habiller le cœur...

1 Introduction

For the project, our idea was to implement a little platform adventure based on the novella by Antoine de Saint-Exupéry.

The player is represented as a little orange fox, freely inspired by the fox in the book. His/her goal is to reach the Petit Prince on the other side of the planet, in this case the Asteroid B-162 of which the prince is originating.
To do so, the fox must collect a certain number of gold coins and avoid the obstacles (red coins and volcanoes, respectively) in the path. The gold coins increment the player's score, while the red coins decreases it. If the fox impacts against one of the volcanoes, the game ends with a *Game Over*.
When the fox has collected 20 coins, the coins and the obstacles cease to appear and the fox can easily reach the prince without any help from the player.

The player can control the movements of the fox using the left and right arrow keys: when pressed, the fox changes accordingly the lane is currently in.

2 Environment

To develop the application we have used the **Three.js** library[1]. We have preferred it over the **WebGL** standard because of a larger possibility of customization on both the environment and the different aspects of the application.
In order to obtain a better performance of the project resources, we have added some libraries connected to the **Three.js** API, but not directly contained in the principal repository.
More precisely, the libraries that we have decided to add explicitly in the *.html* file are:

- **OrbitControls.js**: this library allows an improved control over the camera. We have used it to manage the movements of the scene with respect to the position of the elements[2].
- **OBJLoader.js**: this resource implements a loader that handles the loading of an object: given a set of vertices, it represents them in human-readable 3D images.
- **MTLLoader.js**: this package is a loader for *.mtl* files and is used internally by OBJLoader; that means, even if we didn't explicitly call its methods in the code, it is needed by the other resources. It manages the textures of the object[3].
- **LoaderSupport.js**: this library contains, as the name suggest, a set of support classes for loading resources.
- **CanvasRenderer.js**: this library is used to display the scene and, in particular, to create the renderer; it is an alternative to **WebGL** since it uses **Canvas 2D Context**[4].

3 Technical aspects

3.1 HTML file

The *.html* file can be divided into three parts: the first one contains the style properties, in form of *CSS* commands; the second one contains the *Javascript* libraries we have listed in **Section 2**; the third one contains the *html* code for the page.

3.1.1 The CSS

In the **CSS** part of the file we assign the desired style to the project by handling of the respective properties.
The application has a defined background passed through *background-image*; the initial screen displayed is obtained with a *div* that appears only until the user clicks on **start**;

the score points are presented in a box on the upper left of the screen and updated via *javascript*; when the game is over, another text appears to inform the player that he/she has failed the objective of the game.

3.2 Javascript file

The **init** function is the first to be called. In turn, it calls the *createScene*, *createFox* and *loop* functions, that manage respectively the scene, the fox and the animations.
The **createScene** function creates the scene to be displayed: starts the clock, initializes the renderer and assigns the positions to the camera. Now, the interpreter calls the functions *insertElement*, *addB612*, *addLight*, *addExplosionCoin* and *addExplosionBad*: all of those methods are implemented as auxiliaries to model the layout of the game. Moreover, at this point a listener *onWindowResize* is added in order to manage the resize of the window.
The **insertElement** manages the creation of the bonus and malus: a *for* loop checks whether the application has a sufficient number of coins and volcanoes and, if others are needed, it invokes the *createVolcano*, *createBad*, *createCoin*.
The **createVolcano** implements the creation of the volcanoes, the principal obstacles of the player in the game. From the point of view of the code, they are created as cones through the *Three* class *ConeGeometry*[5]. The aspect of each volcano depends from a texture loaded with a loader implemented through the *Three* class *TextureLoader* and has fixed position in the screen.
The **createBad** is the function that draws the red coins in the game by invoking the *Three* class *CylinderGeometry* and applying a red texture with the *Three* class *TextureLoader*. Then, a fixed position (x,y) is assigned to each coin.
The **createCoin** method essentially repeats the steps of *createBad*: the only difference is that in this case a yellow texture is applied. This is fundamental, since the player must be able to distinguish between the gold and the red coins at each moment.
The **addB612** method is responsible for the creation of the planet. To do such a task, it uses the *SphereGeometry* class from *Three.js*, but we will examine it deeper in **Section 4.1**.
The **addLight** function is responsible for the positioning of the lights. We have inserted an *HemisphereLight*, i.e., a light that illuminates the scene from above and fades gradually as it touches the ground. We have also inserted a *DirectionalLight*[7], that shines from a specific direction as if it was the sunlight from outside[8].
The **addExplosionCoin** function implements the animation of a block pixel explosion when the fox reaches one of the coin: a number of yellow *particles* are created with the same texture as *createCoin*.
The **addExplosionBad** method essentially does the same as *addExplosionCoin*, with the only difference that the references are the *bad* coins.
The **createFox** function implements the logic behind the creation of the fox that the player has to move. We will analyze more deeply how the fox model has been introduced in the project in **Section 4.2**, so let us leave this aspect aside for a moment. The method also manages the dimension, the position and the direction of the fox by operating on

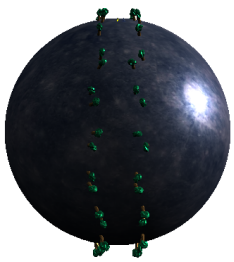
the **mesh** of the fox, obtained as a *Three Object3D*.
The **loop** is the method that checks whether a boolean variable, *gameover*, is false: until it is false, the function calls *animFox*, *updateFox*, *update* and *requestAnimationFrame*. As the name suggests, *gameover* controls whether the player is still untitled to play or not.
The **animFox** method manages the animation of the fox: for each component of the mesh representing the animal, a rotation is computed with the product between the sinusoidal computed on the date, a fixed duration and offset, the *PI* for a fixed amount.
The **updateFox** function manages the switching of the lane by checking when the arrow is pressed throught a *controller* variable, that is handled by an event listener *keyListener*. The change in the position of the fox is carried out controlling in which lane it is currently in and modifying it.
The **update** method handles the rotation of the planet and of all objects; moreover, checks the value of the *clock* in relation with a fixed value (0.5): if it is bigger, then the clock is restarted and the function *addInPath* is called. Furthermore, the functions *checkCollision*, *explosionCoin* and *explosionBad* are called, so to perform the game characteristics.
The **addInPath** is the method that deals with the positioning of the coins. In fact, while the game is running it randomly selects a lane and calls the *addObject* function; otherwise, it sets the variable *finito* to true.
The **addObject** method is called with a parameter, *isPath*, that controls whether the function must create a coin (gold or red) or a tree. In fact, it is called with *isPath* true from *addInPath*, meaning that the object must be called in the lane in which the fox runs; it is called with *isPath* false from *insertBaobab*, meaning that the object is on the left or on the right of the track. If the method creates a tree, then *createBaobab* is called. The object that has been created is then added to the planet.
The **createBaobab** function creates the baobab; we will analyze it in details in **Section 4.3**.
The **checkCollision** function manages the collisions: while the game is running and is not finished, checks the distance between the object and the fox; if the collision happens between the fox and a gold coin, the score is incremented, if happens with a red coin, the score is decremented. The object is then removed from the scene, even if no collision has occurred.
The **explode** function handles the collision between the fox and the object: if the object is a good or a bad coin, then the particles of the explosion are made visible; otherwise, if the object is a volcano, then the *gameover* variable is set to true and the game finishes with the notification to the player.
The **explosionCoin** and the **explosionBad** are essentially the same: the one thing that changes is which object they interact to. The former one controls the appearance of the explosion when the fox collides with a gold coin, the latter controls the appearance of the explosion when the fox collides with a red coin.The pixels that form the explosion are moved outwards.
The **onWindowResize** method is managed by a listener, as we have stated above.

When a *WindowResize* event is registered, the application computes again the aspect of the scene, in height and width, and of the position of the camera. The display is then updated with the new values.
The **setup** function sets up the background of the game and adds all the elements to the scene by the instruction *add* of the *Scene* class: the objects that are inserted in the scene are the *particles*, the *fox*, the *terra* and the *sun*; moreover, it calls the function *insertBaobab*. As the final instruction executed, it initializes the score of the game at 0.
The **insertBaobab** function contains a *for* loop that, until a certain number, generates trees on the left and on the right of the path by calling the *addObject* method with according parameters.
The **createRose** method handles the creation and the visibility of the rose on the scene: we will evaluate how the flower is designed in **Section 4.5**.
The **loadPrince** function, as the name suggest, manages the creation and the display of the Petit Prince on the game. We will analyze it more deeply in **Section 4.5**.
The **insertPrinceAndRose** is an auxiliary function that creates an array with a *loadPrince* variable.

4 Components

As stated variously in the Section above, we now consider the principal components of interest that constitute the heart of the application.

4.1 The planet



The planet represents the principal aspect of the game. From the coding point of view, B612 is a *Mesh* built with a *SphereGeometry* and *MeshStandardMaterial*. The so particular appearance of the planet, chosen bearing in mind that in the novel it is an asteroid, is applied through a texture. The texture is thus loaded with the *TextureLoader* class and is an image in high resolution of the planet Mercury.[10]
The rotation of B612 is the instrument for the advancing of the platform game: as long as

the game continues, in the **update** method the rotation is incremented via *planetSpeed*, a numeric variable declared a priori.

4.2 The baobabs

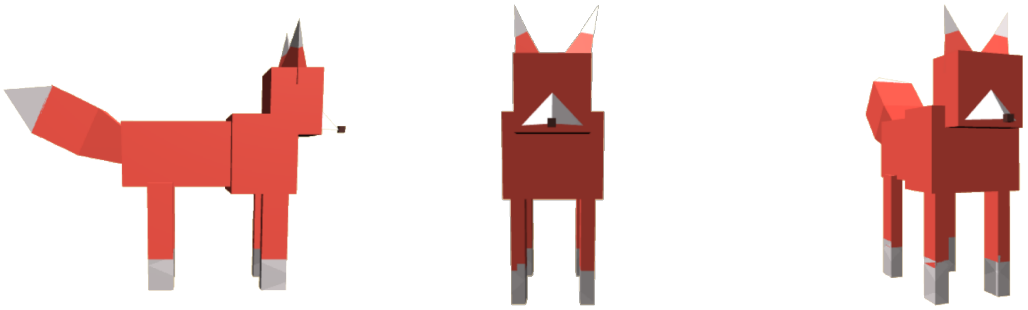
The trees are formed by two compenents: the **top** and the **trunk**.

- the **top** is a *Mesh* composed with *TorusKnotGeometry*, that generates torus geometries[9], and *MeshStandardMaterial* of green color. The rotation for the y axis is random and uses the appropriate *Math* class.
- the **trunk** is a *Mesh* element constituted by *CylinderGeometry* and *MeshStandardMaterial*.

The two items are then added to **baobab**, the *Object 3D* that represents the tree on the scene. Every baobab is added to the scene, both at the left and at the right of the path, using the **insertBaobab** function.

4.3 The fox

The **fox** is a hierarchical model done throught different *Three.js* components. A new *Object3D* is created and the elements that form it are included as *mesh* parts throught a principal one called **mesh**.



The coloration is done through the *MeshPhongMaterial* on a flat shading: the three different colors of the fur, brown, red and white have been declared in an array *Colors*. Depending on which part of the body we are taking into account, a different *furMat* is used.
The components of the fox have been added to the **mesh** starting from the body:

- the **body** is a *Mesh* object created with a *BoxBufferGeometry* and the *MeshBasicMaterial* red. It is directly inserted into the principal block.
- the **chest** is a *Mesh* object obtained with a *BoxBufferGeometry* and the *MeshBasicMaterial* red. It is directly inserted into the principal block.
- the **head** is a *Mesh* object implemented with a *BoxBufferGeometry* and the *MeshPhongMaterial* red. It is directly inserted into the principal block.

  - the **snout** is a *Mesh* object created with a *CylinderGeometry* and *MeshPhongMaterial* white. It depends from the **head**.
  - the **nose** is a *Mesh* object created with a *BoxBufferGeometry* and a *MeshPhongMaterial* brown. It depends from the **head**.
  - the **ears**, respectively left and right, are *CylinderGeometry* objects of red mat, positioned on the opposite sides of the **head**, from which they depend. In fact, only one ear is created via code, the other one is simply cloned from the first one and set with opposite attributes *position* and *rotation*.
    * the **ear tips** are objects composed from *CylinderGeometry* and white *MeshPhongMaterial*. As the **ears**, to which they are added, only one eartip is created via code.

- the **tail** proved to be the most difficult part in the creation of the fox. It is composed via *BoxGeometry*, the basic class of *Three.js*, and then we have modified the values of the vertices in order to obtain the aspect we desired. It is then added to the principal block.
  - the **tail tip** is constructed with a *CylinderGeometry* and a *MeshPhongMaterial* white. It depends from **tail**, to which it is added.

- the **legs** are created as *BoxBufferGeometry* with red *MeshPhongMaterial*. Only one is explicitly created, the three that remain are obtained from the first one via the *clone* function and their positions are set accordingly in order to represent the legs of the fox. They are then added to the principal block.
  - the **feet** are implemented using *BoxBufferGeometry* and white *MeshPhongMaterial*. As we have stated for the **legs**, to which they are added, only one feet is created at coding time, the other ones are realized through the *clone* function and added to the respective **leg**.

The fox animation is handled by the method **animFox**, that takes as parameter the fox and applies to some of its components a rotation with the *sinusoidal* value of the *Math* class. The components that move are the tail (in x, y and z), the head and the legs: in that way, the body of the animal gives the impression of moving.

When speaking of the movement along the path, first we must reaffirm that the structure of the route is formed by three lanes: for the sake of simplicity, we will address them as **left**, **center**, **right**.
The fox changes its path in response to the pressing of the **key arrows**: a *keylistener* function manages the recognition of which key has been pressed, while the function **updateFox** handles the various situations the game could be when the player interacts through a set of *if-else* instructions.

4.4 The coins and the obstacles

As we have stated variously in the report, we have three items that interact with the fox in the path of the game: the **gold coin**, the **red coin** and the **volcano**.

- we will analyze the **gold coin** and the **red coin** together, since they are developed in the same way: in fact, they differ only for the applied texture and in the way the game reacts to the collision with the fox.
  The coins are *Mesh* objects with *CylinderGeometry* and *MeshBasicMaterial*, for the gold coin of yellow color and for the bad coin of red color. Then, a texture is applied with a *TextureLoader*, **goldcoin** and **redcoin** respectively. The coins' position in x and y are fixed, but they appear randomly on the scene: that means that it can happen a scenario with two coins on two lanes at the same time.

- the **volcano** is a *Mesh* object obtained through *ConeGeometry* and *MeshBasicMaterial*. Then, a texture is loaded with a *TextureLoader* to achieve the desired appearance.

If the player impacts against a coin, the score is increased or decreased accordingly to the type of coin; however, if the player impacts against a volcano, then the boolean variable **gameover** is triggered and the game finishes with the loss of the player.

4.5 The prince and the rose

We have decided to use an external model for the **Petit Prince**. It is formed by two components, the model and the material, loaded in the game respectively via the *OBJLoader* and the *MTLLoader* classes.
The prince is then rotated in such a way that faces the fox and is set to invisible until the game finishes.

The **rose**, on the other hand, has been created by code. As for the baobab, we have two components: the **flowerTop**, with *TorusKnotGeometry* and *MeshStandardMaterial* of red color, and the **flowerStem**, with *CylinderGeometry* and *MeshStandardMaterial* of green color. These are then added to the real rose, an *Object3D* that is visible only when the game is finished with the win of the player.

5 User manual

To start the game, you must press on the **Start** screen. The scene will appear: you are a little red fox, and your goal is to collect **20 points** to be allowed to meet the Petit Prince and its rose.
The path is composed by three lanes, you start from the central one and can switch to the left and to the right by pressing the left or right arrow in your keyboard. You obtain the points by striking the **gold coins**, but beware: the route is not smooth, and there are a lot of obstacles.
If you crash against a **volcano**, then it's game over: you will have to retry from the beginning and you will lose the points you have collected. If you bump against a **red coin**, your points will be decreased by one.
At **20 points**, the game stops creating coins and you will soon encounter the Petite Prince and its red rose.
Congratulations, you have won the game.

Et il revint vers le renard:
– Adieu, dit-il. . .
– Adieu, dit le renard. Voici mon secret.
Il est très simple: on ne voit bien qu'avec le cœur.
L'essentiel est invisible pour les yeux.
– L'essentiel est invisible pour les yeux, répéta le petit prince, afin de se souvenir.
– C'est le temps que tu as perdu pour ta rose qui fait ta rose si importante.

References

[1] Ricardo Cabello, *Three.js*, https://threejs.org/

[2] Dustin Pfister, *Quick orbit controls for three.js*, https://dustinpfister.github.io/2018/04/13/threejs-orbit-controls/

[3] Ricardo Cabello, *Three.js docs - MTLLoader*, https://threejs.org/docs/#examples/loaders/MTLLoader

[4] Jos Dirksen, *Three.js Cookbook*, 2015, Packt Publishing Limited.

[5] Ricardo Cabello, *Three.js docs - ConeGeometry*, https://threejs.org/docs/#api/en/geometries/ConeGeometry

[6] Ricardo Cabello, *Three.js docs - HemisphereLight*, https://threejs.org/docs/#api/en/lights/HemisphereLight

[7] Ricardo Cabello, *Three.js docs - DirectionalLight*, https://threejs.org/docs/#api/en/lights/DirectionalLight

[8] Joseph Rex, *Diving into 3D WebGL with Three.js*, https://x-team.com/blog/3d-webgl-threejs/

[9] Wolfram Research, Inc., *Torus*, http://mathworld.wolfram.com/Torus.html

[10] NASA, *Global Map of Mercury*, https://www.nasa.gov/multimedia/imagegallery/image_feature_1614.h