Interactive Graphics: Final Project

Damiano Zappia - 1870958
Gianluca Maselli - 1892187

Academic Year 2019 – 2020

Contents

1 Introduction

The main idea for this project comes from the popular VR game for PS4 called Eagle Flight.
The graphical application developed, consists into a low poly city, with several buildings, a little airport, a Ferris wheel, and many more buildings and objects.
Inside the city you can find several animated things, like cars, rotating Ferris wheel, and an eagle, that is the principal subject of the application. It starting from the tip of the highest skyscraper of the town, flies around all the city making a complex path around, allowing the user to set the view to third person watching the stunning movement of the bird around the buildings, or setting it to first person being totally involved in the fly like if he was himself the eagle.
The principal style of the application is similar to a cartoon one, for this reason the models are easy and intuitive. For the same reason the texture seems to appear quite simple even if a long work has been required.
The eagle follows a defined trajectory and when it finishes, returns to the top of the building where it started to fly.

In order to produce the objects inside the scene, Blender has been used, and it was a good opportunity to develop complex models, stylistically nicer than the one produced by using the normal WebGL, and a the same time, to learn how to use this environment from scratch and become able to build whatever was needed for the project.

2 Environment and Libraries

The models contained in the application Eagle flight are principally made in Blender.
Regarding the city ,in which the eagle performs its flight, both imported models and developed ones by scratch are present.
This choice was made in principle for two reason: the first reason is to maintain a low memory usage in loading the model and consequently to run the application and the second one is to build a model similar to a cartoon style.

On the other hand the eagle is a fully imported model, but modified in the blender environment in order to obtain an hierarchical structure, that is then exploited in the code.

All the Blender models are exported in `.glb` extension due to ease in loading the model in tree.js and, at the same time, to keep the textures correctly attached in all the city structures (this will be explained in the Section 4).

Animations, lights and shadows are defined by using the `tree.js` library.
The libraries used in the application are directly taken from the URL where each library is located in the original tree.js website. This is done in principle to ensure update libraries (avoid deprecation problems), which are:

- `three.module.js`
- `OrbitControls.js`
- `GLTFLoader.js`

3 Hierarchical Models

As said before, the project contains several objects with a hierarchical structure, that is exploited to realize realistic animations. Now we will see in detail all them.

3.1 Eagle model

The main hierarchical model implemented in the program, is the Eagle which has been built by using blender. The model has been firstly imported and then revisited in order to define the hierarchical structure needed for the animations.
The overall structure of the eagle is showed in the figure below in which the names reported are the same used in the blender model: Note that paws of the eagle are built in such a way to replicate the

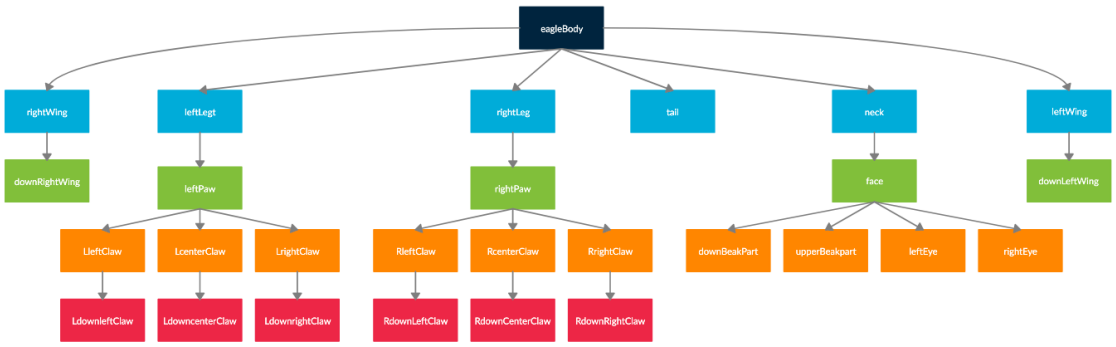


Figure 1: Eagle Hierarchical model graph

natural ones. For this purpose they are divided in tree main parts: the paw itself, the tree upper parts of the claws and the terminal parts of the latter. This structure is the same for both the legs of the eagle.
In the same fashion the wings are divided in two part as well, this is done to build a kind of natural angle formed from the upper part of the wing and the down part of it. This is clearly visible in the animation where the eagle moves its wings according to the realistic bird flight.
Regarding the frontal part, the neck is the part where the face is attached with its relative parts. Note that the beak is made by two part to simulate the natural opening-closing.
The last part is the tail which is a simple single part.

3.2 Ferris Wheel model

The other hierarchical model which is animated even if it is less complicated than the eagle, is the Ferris wheel: Its is an imported model, divided in a similar way of the eagle. The main part in this case is the central axle, that rotating allows to the main ring, linked to the axle thanks to the steel arms, to rotate. Then on the main ring we have a set of pegs, that are the places where each seat is hooked. This division of the seats from the main structure, allows to maintain the correct

orientation and position of each single seat with respect to the wheel while it is moving (we will see how in section 6).

3.3 Cars models

Finally we have the cars. Each car is modelled in blender, it is composed by the chassis and the wheels which are sons of the latter. This is done in order to simulate the wheel movement while the chassis is moving during the animation of the city.

4 Lights, shadows and Textures

The main idea is to replicate the natural aspect characterizing a big city during a sunny day. Thus we introduced all the elements that will make the cartoon city as much realistic as possible, adding lights, shadows, and textures.

4.1 Lights

A `directional light` has been introduced.
This kind of light is the main actor to reproduce the sun light and consequently to create the classic shadow effect showed in the project.
In order to clarify some aspects along with the choices taken a briefly technical explanation is required. To do this the code is reported and explained step by step.
First of all the Directional light is declared by using the `three.js` library, this is easily done by calling `THREE.DirectionalLight`. The color and intensity of the light is passed to the constructor while its position and direction(position of the target) are specified by the two vectors defined by the properties `position` and `target`.
The property `castShadow` is also called in order to create dynamic shadows. This is adopted when an `.glb` models is loaded as well in order to create its own shadows with respect to the light.

4.2 Shadows

As a matter of fact, to obtain a realist effect with the light, as we previously said, the `shadows` must be included. For this purpose is possible to call the property `shadow` which essentially adopts a `DirectionalLightShadow` which is used to calculate shadows for this light.
The `bias` property is utilized to define how much to add or subtract from the normalized depth when deciding whether a surface is in shadow, while the `mapSize` property defines the `width` and the `height` of the shadow map.

The overall work made on lights and shadows is implemented by the following code:

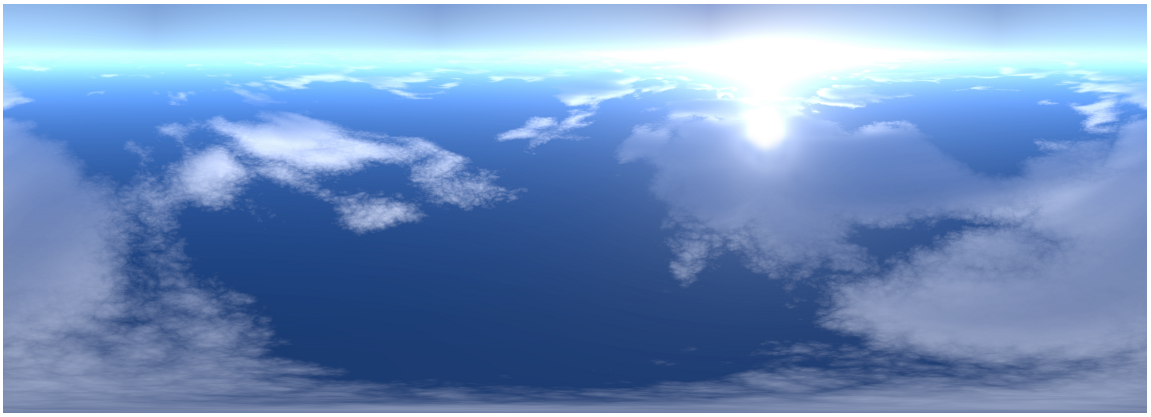
```
1
2     const color = 0xFFFFFF;
3     const intensity = 1;
4     const light = new THREE.DirectionalLight(color, intensity);
5     light.castShadow = true;
6     light.position.set(-1000, 1000, -300);
7     light.target.position.set(-550, 40, -450);
```

```
8
9       light.shadow.bias = -0.004;
10      light.shadow.mapSize.width = 2048;
11      light.shadow.mapSize.height = 2048;
```

Listing 1: Lights Code

4.3 Textures

The other main role to make the scene nicer is the introduction of a `cube texture`. This kind of texture is attached by using a `cube mapping` in which essentially uses the six faces of a cube as the map shape. The texture is divided in six faces starting from a `HDRI texture`:



After divide this texture in six square faces(performed by a tool) and loaded them by using `tree.js` with `THREE.CubeTextureLoader` is possbile to map the texture with the following code lines:

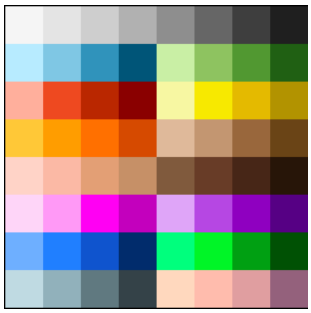
```
1
2       const loader = new THREE.CubeTextureLoader();
3       const texture = loader.load([
4         'sky3/px.png',
5         'sky3/nx.png',
6         'sky3/py.png',
7         'sky3/ny.png',
8         'sky3/pz.png',
9         'sky3/nz.png',
10      ]);
11      scene.background = texture;
```

Listing 2: Texture Loading

Note that each texture is loaded independently and positioned with respect to a face of the cube. This mapping must match with the original texture, otherwise problems will be encountered.

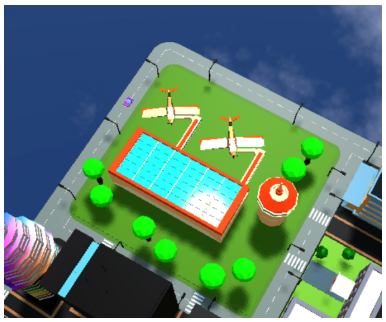
There is an important link between the `directional light` and the `texture`. In fact to keep a realistic effect with the sun, the `light.position` must be the same with the sun in the texture. For the above reason the light is placed a little bit far from the scene.

The part relative to the main scene has been explained above, but it is important to underlying that other `textures` have been utilized.
Indeed, most of the colors present in the model are defined by a single texture which essentially is a palette of colors.



The main advantage of the palette is that is easier to apply in Blender. Regarding the model of the city each face of each building is mapped to a single color contained in the palette. Another important motivation of using the latter is to obtain a kind of cartoon style.

But we also used a color-texture in the airport glass, this can be seen by moving the scene with the mouse, getting the light hitting the glass, that in this case will reflect it, producing a shine effect, as it can be seen below.



The last texture to mention is the one attached to the `eagle`. In this case the model was downloaded and the texture was already attached. Otherwise after dividing the model to obtain an hierarchical structure a new mapping was needed.

Even if the textures are mentioned in this paper they are not included in the project folder. The reason is that by exporting the model from blender in the `.glb` extension, the textures remains internally attached to the object. Then , when a model is loaded with the `GLTFLoader` library the texture will be loaded as well.

5 User interaction

The application presents an homepage which contains two button:

- `Start Flying`: It is the main button to access to the game. It has been built by a simple function which loads the file `.html` referred to the main `.js` file.
  In other words the `homepage.html` calls the `loadcity.html` which loads the file `eagleFlight.js`.
  This solution is the easiest one to switch the page and load the `.js` file by clicking a button.

- `Instructions`: It is an open-close panel which shows the main information regarding the usage of buttons.

After the user clicks on `Start Flying` button, the `.glb` models are loaded and the eagle is located on the top of the highest tower.
At this point the are several keyboard buttons that the user can press to activate different modalities:

- `default`: by default the view is set as overall view over the city, and the user can hold the left button on the mouse, moving it, to change the orientation and see the city from the preferred point of view. By holding also the `shift` button, it's possible to change the position, sliding the camera along a certain direction. This kind of interaction is implemented using the `OrbitControls`.

- `space bar`: By pressing this button on the keyboard the eagle starts its own animation, flying through the buildings. When the animation is terminated, the user can restart it by pressing the same button again.

- `W`: By pressing this button the third person view on the eagle is activated. This view can be exchanged anytime with the first person view, with the following key command. Also,

when the user is in first person view, by pressing `w` again, he can change the camera to return controllable by the mouse.

- `S`: By pressing this button the first person view is activated. Anytime, the user is allowed to switch the first person view with the third person view during the all eagle animation.
- `A`: By pressing this button the eagle scream is emitted. This is implement by using an `audio` (explained in Section 7). Every time the user click this button the scream is emitted.

In the code the keyboard input has been handled in a easy way, by using the method:

```
    document.addEventListener("keydown", onKeyDown, false);
```

The method `addEventListener` set a function which will be called anytime the specified event is delivered to the target element. The parameters passed to this method are:

- `keydown`: The name of the string which represents the name of the event
- `onKeyDown`: It is the name of the function that will be run when the event occurs.
- `false`: When set to false, the event handler is executed in the bubbling phase

The main parameter to discuss is the function that is run when the event occurs, the so called `onKeyDown`. This function is structured in the following way:

```
function onKeyDown(event) {

    var keyCode = event.which;

    if(keyCode == 32){ // press spacebar to make the eagle fly
    startFly = true;
    }

    if(keyCode == 83){ // press S to switch between Orbit Controls and Camera
    active = !active;
    }

    if(keyCode == 87){ // press w to switch between third and first person
    activeCamera = !activeCamera;
    }

    if(keyCode == 65){ // make the eagle shout
    soundEagle.play();
    }

    //requestAnimationFrame(render);
}
```

Listing 3: function for the buttons handling

The first variable called `keyCode` is initialized with `event.which` that contains the key code of the key which was pressed to trigger the event(for istance keydown or keyup).
In fact in each `if` the key code relative to the keyboard button is evaluated, and in the case of a pression, then the code inside it is executed.
As soon as a keyboard button is pressed an event is called. In this case the event set the value of a Boolean variable to execute a code.

This solution was made essentially to keep an animation activated till the Boolean is set to false. To make the things clear lets do an example with the animation relative to eagle.
The first `if(keyCode == 32)` verifies that the pressure of the `space bar` is activated. When this event occurs the variable `startFly` is set to `true`. Therefore the code which will be executed is:

```
if(startFly){
    move_eagle();
}
```

Listing 4: startFly button

This solution is adopted, as previously said, to keep the animation activated. The same thing is repeated for the two view relative to the camera, with the possibility, to switch between them. In the case of the camera the switch is defined as:

```
if(active){
    var relativeCameraOffset = new THREE.Vector3(0,-5, -5);
    var cameraOffset = relativeCameraOffset.applyMatrix4(body.matrixWorld );

    camera.position.x = cameraOffset.x;
    camera.position.y = cameraOffset.y;
    camera.position.z = cameraOffset.z;

    camera.lookAt(body.position);

    if(activeCamera)  {body.add(camera);}
    else  {body.remove(camera); }
}
```

Listing 5: Cameras switch

In this case the Boolean `active` changes the camera from being controlled by user with the mouse, to follow the eagle in the first person view (`s`), while when the Boolean `activeCamera` is used, the third person view is enabled by adding the camera as a son of the body (`w`). To switch between the two cameras continuously the `remove` method has to be used.

Last but not least, the `if(keyCode==65)` is used to make the eagle scream when the `a` button is clicked.

6 Animations

Animations play a key role in this project, indeed we decided to have several animations inside our city, of different complexity. In this chapter we will explain them one by one, focusing in particular on the most articulated one, the eagle flight.

6.1 Cars Animation

First of all we have several cars inside the city, and some of them are animated, making a rectangular path around some of the city blocks. Each car (like the truck, the police car, the red car, ans so on and so forth) does a specific path that is set inside the animation function. This specific path is done passing customized parameters for each car, to the `move-vehicle` function, responsible for the movement. The function receives the specific values for a certain car and actuate the movement

according to them. This allows to have a single function for the movement of several different vehicles, resulting in less lines of code and a better efficiency of it.

6.2 Ferris Wheel animation

The Ferris wheel is a complex model, formed by a hierarchical structure, where the main ring rotates thanks to the arms linking it to the axle. The axle is the parent node of all the model, and we make it rotates around y axis, with a speed that is one half of the flow of time. Then in order to make it possible to have the seats that remain in a position with the seat plan parallel to the ground, while the ring rotates, we act on the structure with a double **for** loop and an **if** statement, that perform a check in the axle's children in order to take all the seats and make them rotates together with the same speed of the wheel, simulated the "gravity effect", with as said the seat plan that maintains the same orientation during all the movement.
The code for this is below.

```
if(axle){
    axle.rotation.y = 0.5*time;
    for (const piece of axle.children){
      for(const peg of piece.children) {
        if(peg.name.includes("SEAT")) {
           peg.rotation.z = 0.5*time + Math.PI;

         }
      }
    }
  }
```

Listing 6: Ferris Wheel rotation implementation

6.3 Eagle animation

The eagle animation is the most complex and articulated movement in the project. It is developed with the utilization of dedicated arrays of values for each specific part of the eagle body.
Before to start explain all the movement, it's better to show the eagle hierarchical model structure, that is the following:

We exploit this complex structure to produce an articulated animation, divided into two parts. A "static" animation where the eagle doesn't translate or rotates its body, but moves the components like paws, beak and wings, and a "dynamic" where the eagle translates and rotates around the city and the buildings.

6.3.1 Static Movement

This movement simulates the flap of the wings. Indeed thanks to the fact that we have each wing divided into upper and lower wing, we can simulate the folding of the wing while the eagle flaps it.

In order to do this, we have a vector called `eagle_moves` that is a bi-dimensional vector, each row of the vector is used to store the values for a specific part pf the eagle to move. The static movement is done using the `lerp()` function, that basically is an interpolation function, that calculates a the numbers between two values that we called `current` and `target`, at a specific increment. The values are stored in a vector that is returned, and put inside a row of the matrix *eagle moves* as

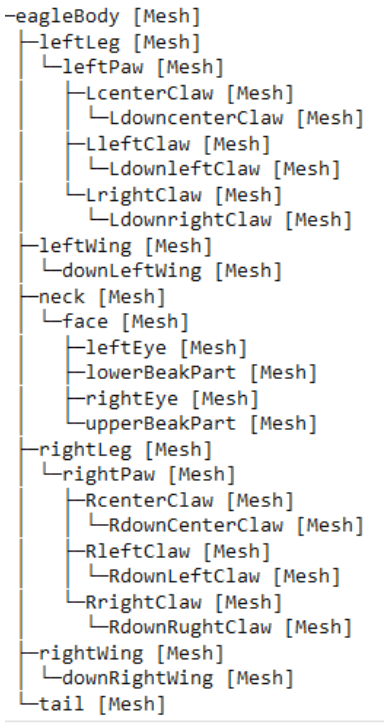


Figure 2: Eagle hierarchical structure

explained.
The code for this is the following:

```
1
2  eagle_moves = [
3  lerp(2,12,wingSpeed).concat(lerp(12,2,wingSpeed)), // for the torso
4  lerp(-Math.PI/4, Math.PI/4, wingSpeed).concat(lerp(Math.PI/4, -Math.PI/4, wingSpeed)
       ), // for the upper wings
5  lerp(Math.PI/4, 0, wingSpeed).concat(lerp(0, Math.PI/4, wingSpeed)), // for the
       lower wings on y
6  lerp(-Math.PI/8, 0, wingSpeed).concat(lerp(0, -Math.PI/8, wingSpeed)), // for the
       lower wings on x
7  lerp(0, -Math.PI/3, wingSpeed).concat(lerp(-Math.PI/3, 0, wingSpeed)), // for the
       paws
8
9
10 ...
11
12
13 //interpolation function
14 function lerp(current, target, fraction){
15
16     var array_of_points = [];
17
18     for (var is = 0; is < (1/fraction); is++){
```

```
      var j = is*fraction;
      array_of_points.push(current*(1-j)+target*j);
    }
  return (array_of_points);
  }
```

Listing 7: Eagle static movement implementation

6.3.2 Dynamic movement

This movement is the most complex of the whole project, and is based on the implementation of *bezier curves*. The reasoning at the basics is the same done for the static movement, where still in the bi-dimensional vector `eagle_moves` we store a row for each of the eagle coordinates position over the 3 axis *xyz*, plus other 3 rows for the eagle rotation around them. Thus in total we have 6 rows on the matrix to control the dynamic movement. The rotation is implemented again with the `lerp()` function, that we use to make the eagle tilt and rotate while it is flying around. Then the three rows used for the translation are used implementing the bezier curves, in particular the *cubic bezier curves*, that are defined over 4 points, and using an interpolation between them creates a curved path with a good level of precision.
The mathematical equation for this is

$$C_0(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3$$

Where `t` varies from 0 to 1, with an incremental amount given by a value called **fraction** that we specify in the code.

An graphical example of what are we talking about is the figure below.

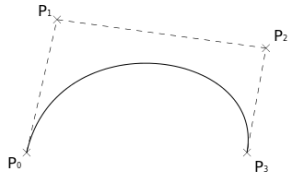


Figure 3: Cubic Bézier curve with four control points

We used cubic Bézier curves to implement a complex movement of the eagle, that starting from the tip of the big skyscraper of the city, when the animation button is triggered, starts to go down towards the ground, and then does a curved trajectory that make it turns towards the stadium, and then continuing around the other buildings, around the Ferris Wheel, and after a while returns to the tip. Thanks to this approach it was possible to have some fancy animations like the **looping**, that is a famous acrobatic movement used in aeronautics, consisting of a complete round, in the form of a ring, done on a fixed plane.

Coming to the code implementation, 28 3-coordinates points were used to create this complex trajectory. The trajetory on each coordinate axis is specified with the following code

```
bezier2(bzX[0], bzX[1], bzX[2], bzX[3], eagleSpeed).concat( // -5-
                bezier2(bzX[3], bzX[4], bzX[5], bzX[6], eagleSpeed),
                bezier2(bzX[6], bzX[7], bzX[8], bzX[9], eagleSpeed),
                bezier2(bzX[9], bzX[10], bzX[11], bzX[12], eagleSpeed),
                bezier2(bzX[12], bzX[13], bzX[14], bzX[15], eagleSpeed),
                bezier2(bzX[15], bzX[16], bzX[17], bzX[18], eagleSpeed/2), //loop
   around the whole city
                bezier2(bzX[18], bzX[19], bzX[20], bzX[21], eagleSpeed/2), //loop
   around the whole city
                bezier2(bzX[21], bzX[21], bzX[23], bzX[24], eagleSpeed),
                bezier2(bzX[24], bzX[25], bzX[26], bzX[27], eagleSpeed)),
```

Listing 8: Bézier trajectory for the x coordinates of the eagle body

Where the bezier cubic function implementation is done in this way:

```
function bezier2(p0, p1, p2, p3, fraction){

  var c_0 = [];

  for (var is = 0; is < (1/fraction); is++){
    var j = is*fraction;
    //l_0.push(p0*(1-j)+p1*j); //L0(t)
    //l_1.push(p1*(1-j)+p2*j); //L1(t)
    //q_0.push( (((1-j)**2)*p0) + (2*(1-j)*j*p1) + ((j**2) *p2))

    c_0.push( p0*(1-j)**3 + 3*((1-j)**2)*j*p1 + 3*(1-j)*(j**2)*p2 + (j**3)*p3 )
  }
  return (c_0);

}
```

Listing 9: Cubic Bézier curves implementation

The choice of the utilization of cubic curves instead of quadric ones, was due to the fact that the cubic curves have a better precision for describing curved paths, and allow also to do "S shaped" paths, that were useful in some parts of the movement.

7 Audio

To make the atmosphere more interesting, audios were implemented.
In total there are four audio tracks with `.mp3` extension:

- `main theme`: Listenable as soon as the `homepage.html` is loaded.
- `city soundtrack`: Listenable as the game start, is the main soundtrack of the application.
- `wind effect`: Listenable as the game starts as for the `city soundtrack`. This two sounds are overlapped even if the `wind effect` has an higher volume than the `city soundtrack`
- `eagle scream`: Listenable every time the user clicks the keyboard button `A`. It overlaps the other two audios, even if it is higher and more intense.

Each audio has been implement by using:

```
var listener = new THREE.AudioListener();
  camera.add( listener );

  // create a global audio source
  sound = new THREE.Audio( listener );
    soundEagle = new THREE.Audio( listener );
    soundWind = new THREE.Audio( listener );

}
```

Listing 10: audio implementation

The first thing to do is to create a new `AudioListener` which represents a virtual listener of the all positional and non-positional effects in the scene.
The listener object is put as a child of the camera (`camera.add(listener)`) in such a way that the 3D transformation of the camera represents the 3d transformation of the listener. At this point is possible to create a non-positional (global) audio object with the `Audio` instruction.

Considering the audio relative to the wind effect, is allowed to load it in the following way:

```
  var audioLoader = new THREE.AudioLoader();
  audioLoader.load( 'sound/wind.mp3', function( buffer ) {
  soundWind.setBuffer( buffer );
  soundWind.setLoop( false );
  soundWind.setVolume( 0.5 );
  soundWind.play();
  });
```

Listing 11: wind audio implementation

The `AudioLoader` class is used for loading an `AudioBuffer` which represents a short audio asset residing in memory, created from, for example an audio file.
The `.mp3` file is loaded by using the method `load` specifing the path where the file is located.
By using the `setBuffer` method the source to the `audioBuffer` is setup. At the same time it set the `sourceType` to `buffer`.
The method `setLoop` is set to `false` because playback should not loop.
Finally the volume of a certain audio is defined by the `setVolume` method. Each audio has a different sound according to the importance given.
In the end, with the method `play` the playback starts.

All the audio are defined in the same way of the `soundWind`, except for the eagle scream, where, how seen in the Section 5, the `play` method is called when the pressure of the keyboard button `A` is activated.

References

[1] Website: https://threejs.org/

[2] Website: https://threejsfundamentals.org/

[3] Slides on "Keyframe animation", treated during the course lectures

[4] Eagle model: https://sketchfab.com/3d-models/sengalang-burong-be3cab654da84e6f8a6ccdc20180a115

[5] Ferris Wheel model: https://www.blendswap.com/blend/19109

[6] Template for the homepage css style: https://www.w3schools.com/w3css/tryw3css_templates_coming_soon.htm

[7] Music and sounds:

- homepage soundtrack: Hans Zimmer - "Homeland"
- main page soundtrack: Eric Buchholz - "Hyrule Field"
- Wind and Scream taken by youtube