Final Interactive Graphics project: ESCAPE ROOM

Antonella Angrisani, Michela Capece
Leonardo Capozzi, Flavia Ferranti

Contents

1 Introduction

One winter evening, by now a long time ago, four friends tried one of the games that has been so depopulated in recent years: the **escape room**. It is in memory of those times that the idea of our project was born.
We are talking about a logic game in which a team of players, locked in a themed room, must find a way to run away using what they find inside and solving codes and puzzles. To successfully complete the game, the escape must be organized within a specific time limit.
The escape that we propose to the player will take place inside a house. He will be made aware of why he is there and, as the game goes on, he will be able to reconstruct the whole story. However, the success of his escape will influence an event that should not be underestimated...
The work was done in `JavaScript`, with the support of the 3D library `Three.js`. We will discuss it in details later, as we use it.
Let's start!

2 Story and instructions for use

It is an autumn evening of the 2000s, a peaceful night on the Lazio coast. You are a *private investigator* and, after a normal day of work, you go to an appointment with one of your clients, *Mr. X*. You have some important news for him. You meet him at 9.00 pm in his beach house. It's a trap!
Who is this gentleman? Why did he decide to hire a private investigator? But, above all, why now did he block you inside his home? These are the questions that grab the player's mind, locked in a bedroom. For now we certainly know the answer to the last question: your ideas on what you have discovered are divergent.

You are in his daughter's bedroom, the light is weak. To move within the game you have to use the mouse to look around and `W A S D` to go forward, left, back and right respectively. Remember, you only have `30 minutes` to get out of here.
To move an object, you have to use `space`. There is a *key under the pillow*, you'd like to try it, even if as you will discover later, it does not open the door to this room. But how to take it? Next to the bed there is a backpack, maybe it's better to exploit it, you can use it to collect everything you find that could be useful for your escape! To get an item, you have to use the `Q` key while to open your backpack and see what is inside, you have to use the `E` key. Choose the object you want to use, among those in the backpack, by pressing the number (between `1 - 4`) associated with it. Some objects must be associated with others in order to work.
Thus, your escape begins: the backpack is the key to unlocking objects. So, you can take the key under the pillow. From now on, you can collect all the items around you that seem important, but be careful never to exceed four of them. For this reason, if you change your mind, you can discard the objects you regretted by choosing the number (between `5 - 8`) associated in the backpack.
On the chair, there is a *rope*; on the ground, there is a *torch*. Damn: it's run out!
Look better around, next to the door there is a *switch*: it moves the *closet* and unlocks a secret passage.

Welcome to the second room! You are now in the parents' room. Go around and find out what is hidden in the furniture. Fortunately, in the drawer of the *bedside table* there is a *battery* for turning on the torch, in the *wardrobe* a *safe*.
It's time for you to know another piece of the story. In 1974, Mr. X married a young woman with whom he was madly in love and from this love, 6 years later, a beautiful baby girl was born. Their lives changed when his wife died in a car accident: their daughter was only 3 years old.
These are the three most significant dates of his life. These numbers contain the correct combination to open the safe. You have to mix numbers of the two years that gave him the most important people in his life. The combination is in fact `7480`, which associates the marriage and the birth of the young girl. Inside the safe,

there are a *diamond* and a *music sheet*. Why keep the latter so secretly?
Think carefully about what you've collected. The key you found in the previous room opens this door.

Welcome to the hallway! There is another piece of furniture. In the drawer there is a *gun*, but there are no shots inside.
No block to access the next room. But are you sure you have got everything you need? From this moment on, you no longer have the opportunity to go back.
Here is another piece of story for you: after much researches, you discovered the identity of the man who caused the accident in which Mr. X lost the love of his life. He wants revenge: that is why he asked for your help! Your ideas of justice are different. You are a law lover, he wants to do on his way. You must reach him in time, you must prevent him to commit a crime!

Welcome to the living room! You're almost there. Look around, you have two escape routes, you just have to understand how to unlock them. The criterion is always the same: choose the right object that will help you. There is an old *violin*: the music sheet found in the safe has to be performed by this instrument.
The only bullet you have available is stuck between the sofa cushions. Maybe has Mr. X lost it going out?
Turn the *mirror*, look for clues in the *bookshelf*. Shaking the latter, a book which contains a photo falls down: it is a classic to keep a photo you are attached to in a book. It is so beautiful, it is from their wedding. But look at it carefully, why reduce it to this state? United on their wedding day, but now separated from life. What will it mean? When they unite themselves, they separate... What is the object that will unlock the exit? The scissors that physically cut this photo or the time that separates them from their happy moments? Which object among the *scissors* on the large table, the *pocket watch* on the stool and *hourglass* on the table in front of the sofa is the clue referring to? The correct answer is: `scissors`.
However, both the scissors and the hourglass unlock the exits: the *main door* is associated with the first one, the *window doors* with the other. The right choice will give you an advantage in the next step, the last one.

Welcome to the garden! Pay attention, the gate is guarded by a big dog. You can make it fall asleep with the sweet melody of the violin. The benefit you may have gained is a longer monster sleeping time.
But don't fool yourself, it's not so easy: you need something to open the gate!
Discover the clues in the garden: go to the table first, pick the rose on the bench and hose down a little bit the garden. Now, you have to look for the one who is clumsy, short and keeper of treasures: maybe he has yours too... Reach the gnome, move it and you will find the key.
You are really almost done. Reach the exit, but don't forget to play the violin. Insert the key: you won!

3 Creation of the house

One of the most important points of our escape room is the scenography, in which the whole game is developed.
The game takes place inside the house of our character Mr. X and now we will explain how it was built.
In order to create the home structure, `GridHelper` was very helful, a 3D object of `Three.js` that defines grid which allowed us to understand the position of the components of the house. We have proceeded starting by putting first the floor and then we arranged the grid inside the scene looking for the right position in which to insert our element. Once founded, we positioned the wall by assigning it the coordinates understood with the grid. The same reasoning was used also to place all the other walls and the roof. Moreover, the procedure was applied to all the other rooms. When the whole house was completed, `GridHelper` was removed because no more necessary.
Each room was created with a function which takes in input the size of the room. The floor is a *plane* implemented thanks to the help of our function `createPlane`. The latter takes in input width, height, the

vector position, the vector rotation, used to place and orient the object, and the material of the plane, which includes textures. The plane is generated thanks to `PlaneGeometry` of `Three.js`, which, given the width and height, builds a regular plane.
The roof and the walls instead are *shape* implemented with `createShape` which, similarly as before, takes in input the initial point of the element, height, width, the vector position, the vector rotation, the materials and possible holes in the walls for doors and windows. The *shape* is generated thanks to `ShapeGeomentry` of `Three.js` which allows to create an irregular plane.
The above mentioned holes, was created specifying width and height of them plus the initial point on x and y axes for positioning them in the desired location of the wall.
As said, all the elements have texture applied. The texture are loaded with `TextureLoader` of `Three.js`. They are wrapped horizontally and vertically and then repeated several times along the entire surface.
Both `PlaneGeometry` and `ShapeGeometry` are created as `mesh` which takes the plane and the material associated. At the beginning we had used the default material `MeshBasicMaterial` but this one is not affected by lights and so it has been replaced with the `MeshPhysicalMaterial`. For more information, the material topic will be expanded in the lights and materials' paragraph.
The scenography is developed on a hierarchical model. The root is the *scene* defined as `Scene`, which allows to decide what and where is to be rendered by `Three.js`. It has as children all the rooms. Also them are built as hierarchical models. In fact, each room is a group, and each element of the structure is added to it. We can see the group as the root of the hierarchical model and each wall, floor, roof but also light and objects as children of the root.

The house is obviously furnished and for doing that we added the objects in the rooms.
In this project, we employed two different types of objects: `.obj` and `.gltf`.
For loading an `.obj` resource, the libraries `MTLLoader` and `OBJLoader` were used. We started with `MTLLoader` for the `.mtl` file, which contains the material of the object that were added on the `OBJLoader` and passed to the `.obj` file.
Since the `OBJ` file format is very old and it provides no scene graph and furthermore everything loaded is one large mesh, we decided to import most of the elements in `gLTF` format.
The latter was designed for import models and display them with a minimum of problems, in a way that the other formats have never been able to do.
Once the object is loaded, we can place it inside the room, scale it to give the right proportions and, if necessary, rotate it and then it can be added to the group.
To develop the actions in our escape room we apply animation on our objects. The objects are themselves hierarchical models composed by the root, which is the main component of each of them, and all its parts are children of the root. In this way it was easy to animate the different components of the object as we will see later on.

4 Sequential house loading and removing

In order to have a fast working project we decided to make the house, and so all its internal models, to load one at the time and to remove the things that, after a while, were not needed anymore.
To do that, we've decided to link the loading of a certain room with the action that underline the passing of the level.
Of course the skeleton of the house is the first thing that is loaded and the only thing that remain during the entire course of the game. At the same time we have placed the girl's room, where everything begin. Once the player manages to understand how to reach the second room (so when it moves the wardrobe entering in the second level) the bedroom is created and so the game can continue.
The same logic is implemented while passing through the parent's door and arriving in the hallway.

From now on, other then the loading, the removing is also implemented. Arriving to the entrance of the living room, the player is asked to be sure of the decision to go on, because it will not be able to go back and take items that may be useful. If he decides to continue the game in the living room, then the view and the movement of the user will be controlled so that he will walk on the opening door. At the same time, behind him, the hallway and the bedrooms are removed from the scene, with the function `removeRooms`, so that the living room can be loaded fast and without problems. The player will be able to gain the control of the motions only after everything is done and, if he decides to look back, he will not be able to open that door again.
This method is implemented also while arriving to the last level, as to say to the garden. Here the player has the possibility to go out from two different doors, so the loading of the garden take into account this possibility. Although the rest remain the same. Once opening one of the two door, he will be guided out the room while the last piece of the house's internal structure is removed from the scene and the garden can came to life.

5 Materials and Lights

In the house light is an important factor. We want to create a dark and disquieting scene to instill in the player a sense of fear due to the situation.
As mentioned before, at the beginning we chose the `MeshBasicMaterial` that has been replaced by the `MeshPhysicalMaterial`. This mesh material has a different approach from the older one. Instead of using approximations for the way in which light interacts with a surface, a physically correct model is used. This type of material was created with the idea that each element has to react correctly under all different types of lights. It follows that `MeshPysicalMaterial` gives a result more realistic than `MeshLambertMaterial`, which is indicate for non-shiny surface and gives no specular highlights, or `MeshPhongMaterial`, which is indicate for shiny surface and gives specular highlights, but it is more computationally expensive.
`MeshPhysicalMaterial` computes shading as `MeshPhongMaterial` does. The shading are calculated per pixel in the fragment shader which gives an accurate result.
The first light implemented is the *ambient light*. As its name suggests it is the light of the whole ambient and it is defined by a color and by a float number which is its intensity. In our case we decided to use a dark gray with low intensity equal to 2.
The other type of light implemented is the spotlight. This one is implemented in the chandeliers and to simulate the light of a torch. For the chandelier, the spotlight is vertical and it is situated into it; for the torch, it is horizontal and positioned at the level of the camera. In this way, we want to simulate the player which has the torch on in the hand.
The spotlight has many parameters:

- *color* and *intensity* (as the ambient light);
- *position* and *target position* : the spotlight points from its location to the one of the target;
- *angle*: the maximum extent of the spotlight;
- *distance*: maximum range of the of the light;
- *penumbra* : attenuates the spotlight cone;
- *decay*: allows to the amount of light to decrease when the distance from the light source increases;
- *castShadow* : if it's true, then the light casts dynamic shadows;

Since the objects in the scene are illuminated, we need to implement `castShadow` on all them in the way that they can project their shadows and this makes the scene as real as possible.

6 Player movement

In order to handle the movement of the player, as if it was actually inside the house, we decided to use the `Three.js` library `Pointer Lock Controls`. This library allows to handle the first person movements basing them on the ones of the mouse that changes over time. It very useful because it gives you access to raw mouse movement, locks the target of mouse events to a single element, eliminates limits on how far mouse movement can go in a single direction, and removes the cursor from view. Pointer lock provided to be very helpful because, once the pointer is locked it is not released until an explicit API and so the rotation implemented through it can be done by simply moving the mouse without clicking any button or drag it around the screen. This allows our player to rotate and look around changing the point of view very easily in order to look for clues and items. The association between mouse and camera ends when the player click `ESC` and pause the game and it's restarted when the player click the mouse button on the *canvas*. Furthermore, he is able to control the mouse also when he loses (due to time ending or to the proximity to non-sleeping dog) or when he wins; having also the possibility to restart the game.
Since the cursor is hide, we thought about adding a marker to make easier for the player to decides and understand better where to go.
At this time the only thing that misses is the possibility for the player to move back, forward and left or right. In order to do that we created the function `listenForPlayerMovement` that takes in input the moving choice of the player. If the player clicks `WASD` the camera goes respectively forward, left, backward and right and when the button on the keyboard is released the movement stops. The result of this keyboard listener is used inside the function `animatePlayer` that associates to the player a velocity vector that applies the movement along a direction vector. Here the only possible directions will be along x and z-axes, because the player doesn't need to jump or anything (for this reason the y view is stuck at a certain value: the height of the player).
The actual movement is done by increasing or decreasing the values of this vector by a number that depends on the player-speed, in our case 200, and a value *delta* that keeps track of the temporal variation needed for the rendering of new frames.
At the end, we apply to the camera the variation calculated so that the player can move.

6.1 Player Collision

The only problem now is that the player can still go everywhere, both insides objects and walls. To avoid this behaviour, we used `raycaster` to detect a collision between the player and every object. `Raycaster` is very useful because it casts from the desired origin point, in this case the camera, a ray that goes in the specified direction. If this ray intersects one of the things putted inside a specified array, then this interaction is pointed out along with the distance to that object.
To implement this behaviour, we created a specific function: `detectPlayerCollision`.
In the `collidableObjects` array we put everything that the player cannot cross: the house structure and the furniture of each room, by also taking every single child of every model.
In order to detect every possible collision, the direction of raycaster must be modified according to the direction in which the player moves. So listening to the pressing on the keyboard, the direction of the camera is rotated according to the right angle.
In conclusion, if this ray intersects with an object and the distance is less than 5 then a flag, set to true, is returned and the movement of the player is stopped.
Furthermore, in order to avoid that the user could pass over the 3D models, we decided to add a *camera body*. This body actually correspond to a transparent cube positioned under the camera. In fact, the location and the quarterion are copied from it for exception of the y values that is lower. This artefact was needed so that, when the player moves, it carries around this cube positioned at his feet.
We look for the collision even for this cube, so that when the cube meets another object we avoid that the

user goes on it. In order to do that, the function `detectedCameraBodyCollision` was ideated. Here we define a matrix where the values are extracted from the rotation of the cube position. Later on we create a vector, with coordinates (0, 0, 1), so that we can apply to it the previously defined matrix (to associate it to the camera body) and the quaternion of the camera itself (to make possible for the vector to change according to the rotation of the user's mouse). At this point we can instantiate another `raycaster` object created with the cube position and this defined vector.
The principle of collision is the same as before calculated with the distance from the ray to the object, even if here the distance is larger than the one used for the player itself.
The function is called as support for `detectPlayerCollision`. In fact, we verify this adding collision after the one of the player and only if the player wants to go forward. So if a collision is detected the user can still move in the other direction or point to a place with no objects.
At last, in order to avoid to the user to exit in some way from the rooms where the game take place, we also defined a series of coordinates to indicates the limits of the camera position. These coordinates are defined thanks to two vectors that express the initial x and y coordinate and the final one. At this point every time that the player moves, we check if the position of the camera is inside this square of coordinates, otherwise the user is moved back inside the room.

7 Animations

As mentioned earlier, animations have been applied to the *hierarchical models*. Each object has a specific goal and the user can interact with most of them. We could make a distinction between the things: some contain the features of the game; others have the purpose of interior design and the user interacts with them to move inside the house.
The logic behind the interactions that the player can do is similar to the one of the collision. We decided to create another `raycaster` object that starts directly from the marker and strikes the objects that must be animated and so that where put in a specific array called `objectsRaycaster`. If the distance from the user and an object is in a specific range, than the animation with that specific object can start. Once the animation starts it can't be stopped, even if the distance from the object change or the model is no more pointed from the player. The user can distinguish between normal models and animated one because, if he is in the right position the instruction to start the movements are displayed on screen. Once all the possible animation associated to an object gets performed by the player, the object gets removed from the array so that the interaction cannot be done again.

7.1 Backpack and animations on objects

The backpack is the first object that must be taken: it permits to collect items scattered around the house. In this way, the player can collect them and use a specific thing in a specific moment.
Its creation is defined in a `JavaScript` class: the constructor defines the object's properties; setter and getter methods permit possible interactions with instance variables; the other methods and functions determinate the general backpack behavior.
The instance variables that define the class are: an *array* which store the objects collected by the player, an *integer* which keeps track of the elements number into the bag and a *boolean* one to know if it is is open or not.
This element allows the following operations: *collect*, *use* and *discard* an object. To describe them, we have specific methods for each action.
Established the backpack, the other elements with whom the player can interact are defined by another class as follows:

- `object`, to keep the reference with object added to the scene;

- `animation`, to store a function which executes the action of the current element;
- `reverseAnimation`, to store the inverse action declared in the animation function;
- `coditionedAnimated`, to know if the current element needs another object to execute its action (boolean variable);
- `isElemOfBackpack`, to define if the object can be collect or not by the user;
- `subjectAction`, to store the name of the object that needs the current element to perform its action;
- `subjectMerge`, to keep the reference to the complementary object of the current element;
- `valueMerge`, in case of merging between two objects, to know which one incorporates the other.

7.1.1 Insert element

This action is managed by the `insert` function, declared in `backpack` class. This function takes as parameter the object to be collected. Before being stored correctly, a single object is subjected to some checks. The first one verifies if the array is filled and, in this case, the object will not be inserted. After that, we have to control that the `subjectMerge` property of the item is different from null: if the condition is equal to false, the object is correctly included; otherwise, we look for the complementary object. If the latter is not contained in the array, the input instance is added normally; otherwise, the two objects are merged together, occupying a single place into the backpack and only the object with `valueMerge` property equal to true will be visible to the user in the graphic interface. When an object is inserted into this array, the corresponding icon is added inside the HTML.

7.1.2 Use object

To use an object, the backpack must be opened. Once the player opens it, he has the possibility to choose the item he wants pressing a number key from `1` to `4`. The number corresponds to a specific position into the bag. In this way, the function `useObject`, declared in the backpack class, is called.
This function takes two parameters: a number, to indicates the index into the backpack's array, and an object, to identify the current element pointed.
The animation is executed only if determinate checks are proved. The first one verifies if the input index corresponds to a null object: in this case, the player is notified with a graphic interface and the animation is not executed. Then, the object is subjected to the following `else if` conditions:

- the `subjectAction` property of an object is different from null: this means that the object is used to execute the animation of another, the one with the same name stored in `subjectAction`;
- the object name compared with the one of the `subjectAction` : if this name is equal to the second one then we call the function `executeAction`, defined in the backpack class, otherwise an alert is shown to indicate that this object cannot be used. Notice that the object parameter, previously mentioned, is used only in this method;
- `subjectMerge` property of the object is different from null: this means that the object that the player wants to use cannot be selected. Since the union with its complementary object misses, the object is not considered completed. Also in this case, we will show an explaining alert;
- in all other cases, we delegate the animation of the object to the `executeAnimation` function.

7.1.3 Discard object

Discarding object allows to free a position inside the schoolbag. Also this action has been implemented by a function declared in backpack class. To discard an object, the user must press a numeric key from `5` to `8`, obviously with the backpack open; otherwise, the action will not have any effect.
This function takes a numeric parameter, that indicates the index of the object to discard. So, we remove the object from the array and the corresponding icon on `HTML`. If the index corresponds to a null object, we show an alert to the user to inform him that he did not select any object. Moreover, for specific objects, like torch, if the action of the object is running, we reset it too.

7.2 Execute animation of an object

As previously mentioned, an object is a class. This class is called `Thing` and, other than containing the instance variables of our objects, it also owns the animation stored in a specific one. This function takes two parameters: one is a numeric value, to define the time of animation and the other is a boolean value, to indicate if you can execute the animation or not. Inside the function there are the following `else if` conditions:

- `isElemOfBackpack` is true: in this case, the object with which the user interacts is an element of the backpack and it will be deleted from the scene, to simulate the collection of the object;
- `flagDoubleAction` is true: the variable subjected at the condition is boolean and indicates if it needs to execute the animation or the reverse one of the object; the latter is executed only if the variable is true;
- animation instance variable is not null, so we execute the animation of the object. There are two different types of animation: one takes the same parameters declared in the `executeAnimation` function of the `Thing` class, the other has no parameters;
- in all other cases, return false;

Notice that an element of the backpack cannot call its own animation: calling `executeAnimation` function and having `isElemOfBackpack` equal to true would always generate the execution of the first condition. This situation is managed in the `useObject` function of backpack class: when an object is used, the `isElemOfBackpack` variable is set to false so that we can allow to perform its animation.

7.3 Interpolation

So far we have explained everything about objects. Let's not forget the player is in the house and, in order to escape, he must move around and collect the objects that are inside the furniture. Therefore, he needs to move things, open drawers, wardrobe, doors and shutters. All these actions have been implemented with the **interpolation**.
The goal is to move the object from an initial point until it reaches the desired position. Given an interval of time Δ`t` and a timestep `t`, the function evaluates the points of a line between x_0 and x_1:

$$x = x_0 + \frac{t-t_0}{t_1-t_0}(x_1 - x_0)$$

Interpolations are necessary, because animations without them appear unstable.
Since the models are hierarchical, the animations are applied not always on the whole `root`, but sometimes only on its `children`.
This reasoning is visible already in the first room: on the bed, the switch and the wardrobe. Let's analyze these animations individually. After identifying the singular components of the bed, we applied the interpolation function only to one of its children, the *pillow*: we press `space` and it goes down.

The other two animations are connected: as soon as we turn on the *switch*, a rotation is made on it and the *wardrobe* starts to move on its x-axis. Also for the switch, we have found the correct component, while the closet moves entirely.
All the animations of which we talk about in this paragraph are defined inside a variable `animation` within the object itself, or at most within the object whose animation is associated. `Animation` is an arrow function, which takes as input an incremental value `t` and a boolean one.
In the second room, we can open a *drawer*, the *doors* of both the wardrobe and the safe. After identifying the children of the objects, on the doors are applied rotations around the z-axis and on the drawer a translation on y. On the latter, is also implemented a reverse animation, which allows us to close the drawer (pressing `space` again). Finally, to exit the room, thanks to the insertion of the key, the user opens the door, which performs another rotation.
In the hallway, we again have a furniture with a drawer and a door. In the living room, the animations are implemented in order on the *mirror* and on the *bookcase*, as well as on the two exit doors. The rotation of the mirror around its y-axis is associated with a clue for the reader, as it also happens for the library shaking on the z-axis.
The opening of each door is associated with a particular object. Moving the hourglass allows you to unlock rotation on the y-axis of the window doors; acquiring the scissors allows the main door to rotate on the z-axis. From the choice of the door comes an advantage for the user: the violin plays for more or less time.
A similar sequence of actions has been implemented in the garden. So, the players needs to proceed for steps. To win, it is necessary to take the key hidden under the *gnome*. However, we cannot move it until we have read all the clues associated to various objects in the garden. The player has to go to the *park table* first, then proceed to take the *rose*. To simulate the collection action, a rotation around the z-axis was applied to it. At this point the *watering* is unlocked: with a translation on y-axis first and a rotation on x later, to simulate its real use. Finally, we can move the gnome, take the *key* associated with the gate and run away.
The fundamental object in the living room, for the purpose of win, is the violin. In fact, if the latter is played near the monster, it manages to make him fall asleep with its sweet melody. In this time interval, equipped with a key, the player must open the gate: on both doors, a translation on the x-axis is applied.
Finally, let's focus on the movement of the *monster*. It continuously walks in front of the gate. Its movement is handled within the function `animateMonster`, which manages the alternation of a translation on the z-axis and a rotation on the y-axis. Inside it, the two functions `moveArms` and `moveLegs` are called for the movement of the four paws, always exploiting the principles of the interpolating function.

8 Audio effects

In the game we implemented some sound effects. For instance, we have the one of the creaking of the door while opening or of the flashlight click. To add these motives, we used the `Three.js` documentation, in which there is a dedicate section for sounds integration. We created a listener, that is an instance of the `AudioListener` class. The latter represents a virtual listener of all the positional and non-positional audio effects in the scene. The single sound is defined by an Audio component, in fact we created an instance of Audio for every effects added to the game, and we passed to it the listener like a constructor parameter. After that, we created an `AudioLoader` in which we loaded the sound with `.ogg` extension. During the definition of the load function, we specified the parameters of interest of our sound, for example the volume. Following this method, we defined all sounds that can be played and stopped, through `play` and `stop` methods, in a specific moment, for example during an animation.

9 Appendix

Here we least all the support, mentioned above, used to define the project

- Three JS
- GLTFLoader
- OBJLoader
- MTLLoader
- PointerLockControls