In the .html file I commented the lines of code to explain the single blocks of code. This is a summary of the techniques used.

1 Game Manual

The game is called "Neon Maze" and its purpose is to reach the exit of the maze by avoiding all the obstacles in the path. The player has only 3 lives to complete the run (a run is the period of the game between two game overs) and win the game. When the player loses a life (it happens when the player hits an obstacle), he returns to the starting position with a life less. There are 3 possible starting positions in the maze and each run has a different starting position. If the player loses all the lives he obtains a game over, he is brought back to the starting position and a new run starts. The player can find 2 kind of perks that can help him in reaching the exit:

1- A "heart perk" that gives him back a single life.

2- An "invincibility perk" that makes the player invincible for 10 seconds during which he can pass through the obstacles without losing lives.

To win the game the player has to hit the ladders that can be found in the central room of the maze.

Commands:

Move the player: 'W-A-S-D' keys or the keyboard arrow keys.

Jump: 'space' key.

Slide: 'shift' key.

Open the menu: 'esc' key.

Start/Restart the game from the menu: click with the mouse anywhere on the screen.

2 Environment, External Libraries, Models, etc

To create the game I have only used 'three.js' and no other external libraries. All the hierarchical models have been created with 'three.js' and the animations have been created by changing the rotation value and the position of the hierarchical models' parts. The only external things used are:

1- The images for the textures, that I found on internet.

2- The heart-shaped meshes that represent the lives of the player, that I found here: "$https://threejsfundamentals.org/threejs/lessons/threejs-primitives.html$.

3- How the movement of the player can be recognized (the events keyDown and keyUp and the usage of the moveForward-Backward-Right-Left variables), that I found on the here: "$https://threejs.org/examples/\#misc_controls_pointerlock$".

3 Scenes

I used three different scenes to create this game:

1- The "menuScene" that contains the writings representing the menu screen and the victory screen.

2- The "overlayScene" that contains the polygon meshes representing the lives of the player, a background plane for these lives and the alerts after a loss of a life or a game over.

3- The main "scene" that contains all the hierarchical models, like the player, the maze (floor,walls,ceiling), the lights and the obstacles.

The change of the scene is due to different flags: If the "menuFlag" flag variable is true, the "menuScene" is rendered. Otherwise, if the "first" flag variable is true then the main "scene" is rendered, and over it the "overlayScene" is rendered. Otherwise, if the "firstPerson" flag is false then the main "scene" is rendered with the third person camera, else, it is rendered with the first person camera. Over the main "scene" the "overlayScene" is always rendered.

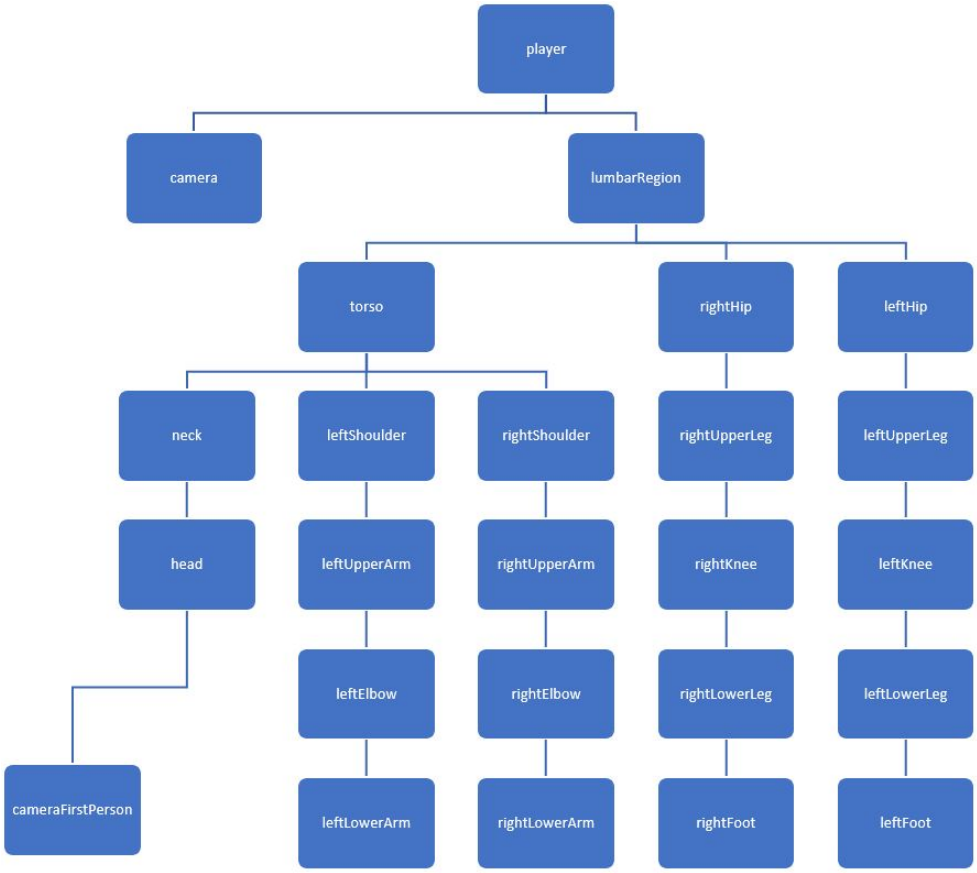
The "first" flag becomes true when the user presses the key 1 and becomes false when the user presses the key 2. The "firstPerson" flag becomes true when the third person camera collides with a maze's wall.

4 Lights and Cameras

The "overlayScene" and the "menuScene" have a white Directional light placed in position (0,0,100) to illuminate the alerts and the objects. The main "scene" has a white Ambient light to illuminate the darker parts of the maze, and a good number of white Point lights put to better illuminate the environment. There are 3 cameras, 2 "Perspective" cameras used for the main "scene" and 1 "Orthographic" camera used for the "menuScene" and the "overlayScene".

5 Player model, Third/First person camera

The hierarchical model of the player has the following structure:



All the dimensions of the body parts depend on the torso's dimensions, so that the body is proportionate. Each part of the player is created with a "BoxGeometry" (with its dimensions) for the geometry and a "MeshBasicMaterial" (with a different texture to create the player figure) for the material. I used this kind of material instead of a "MeshPhongMaterial" because I wanted to have the player always visible and illuminated. Obviously,

the position of each part of the body is related to the origin of the parent (which is the center of the parent object). The third person camera is a child of the player and it is put slightly behind the head and it is looking at the position of the "lumbarRegion". The parameters of the third person camera are: 'fov' equal to 75, 'aspect' equal to width/height of the window, 'near' equal to 0.1 and 'far' equal to 1000. The first person camera is a child of the head and it is put slightly ahead the head. Its parameters are the same of the third person camera, except for the 'fov' value that is equal to 90 (in order to have a better view of the environment). Both of them are "Perspective" cameras. All the joints (neck,shoulders,elbows,hips,knees) are used to simulate a natural rotation of the biggest player body's parts (head,upperArms,lowerArms,upperLegs,lowerLegs). The "lumbarRegion" is used to allow the player model to move, in different ways, the legs and the upper part of the body. In this report I called:

- "upper part of the body": the right/left Hip and the right/left Shoulder.

- "lower part of the body": the the right/left Knee and the right/left Elbow

- "resting position": Is the initial position of the player's model in which all the rotation values are equal to 0 radians.

6 Creation of the maze, of the obstacles and of the lights

The maze is represented as a matrix of dimensions 27x31 (rows x columns), in which each element represents a different object. The floor and the ceiling of the maze are represented through 2 different planes that are Meshes with the same "PlaneBufferGeometry" as geometry and "MeshPhongMaterial" (with a repeated texture, created by using the "RepeatWrapping" and repeated for (15,50)) as material. The floor is rotated of -90 degrees around the x plane to face the player and the ceiling is rotated of 90 degrees around the x plane. In order to avoid the overlap between the floor and the player's feet, the floor is put slightly below the player's feet (y = -0.3 the floor and y = 0 the player).
To read the matrix, that represents the maze, I used 2 'for' cycles: without the obstacles, the position in the space of each element in the matrix corresponds to the values: [(i - rowsCenter) * wallBase] for the position on the z axis and [(j - colsCenter) * wallBase] for the position on the x axis. The 'rowsCenter' and 'colsCenter' are the central elements of the rows and the columns. The wallBase is the height and the width of an ideal square that represents the occupied space of the element and i and j are the values of the 2 'for' cycles. The position on the y axis depends on the different object.

In detail, the list of the elements (all added as children of the main 'scene'):

- '0.1': It represents a possible spawn position of the player. The spawn position is determined at the beginning of each run and every time the player loses a life, he is brought back to the spawn position. In order to determine it, I used an auxiliary function called "setSpawnPosition()" that at the beginning of each run ("endGame" variable equals to 0 [see the next sections]) randomly chooses the new starting position. The coordinates of the new starting position are stored in 3 different variables called 'xPos','yPos','zPos'.

- '1': It represents a maze's wall made of a Mesh with "boxGeometry" as geometry and a "MeshPhongMaterial" (with a texture) as material.

- '2' and '3': They represent horizontal cylinders, placed near the floor (the player has to jump them not to lose a life), with spikes on them. The value 2 identifies the ones that rotate around the x axis while the value 3 identifies the ones that rotate around the z axis. To create the hierarchical model of this obstacle I used the function "newCylinder()": it creates a Mesh representing the cylinder by using a "CylinderGeometry" as geometry and a "MeshBasicMaterial" (with a texture) as material (all the obstacles have been created with "MeshBasicMaterial", because I wanted them to be very visible). All the spikes have been added as children of the cylinder. Each one of them is represented by a Mesh

with "ConeBufferGeometry" as geometry and "MeshBasicMaterial" (with a texture) as material. There are 2 rows of spikes and in each row there is a spike for each 'side' of the cylinder (0,90,180,270 degrees). Finally the cylinder has been added as child of the main "scene" and the function returns the cylinder object.

- '4' and '5': They represent the same horizontal cylinders of the previous element but they are placed higher (the player has to slide them not to lose a life).

- '6' and '7': They represent two cylinders put on the same line that move with different senses (if one moves towards the other one moves backwards and vice-versa). The value 6 represents two cylinders that move on the z axis and the value 7 represents two cylinders that move on the x axis. The position of these two obstacles is not aligned with the center of the matrix's element but it is slightly shifted on the axis in which they move, to allow them not to overlap during the animation.

- '8' and '9': They represent an horizontal cylinder moving on the y axis. The value 8 identifies the ones that rotate around the x axis while the value 9 identifies the ones that rotate around the z axis. Its initial position depends on the height of the maze's walls.

- '10': It represents a plane of spikes that comes out of the floor. To create this object I used the function "newSpikePlane(list[])" that creates 12 spikes (equal to the ones that I created for the cylinders) organized in 3 rows (4 spikes for each row) and returns the plane of spikes object. To warn the player about this obstacle I put on the plane of spikes a number of circles (with the same texture of the spikes) equal to the number of the created spikes. Each circle has a "CircleBufferGeometry" as geometry and the same material of the spikes. All of them are added to the input list of the "newSpikePlane(list[])" function. After the function has been performed, all the circles in the list have been added to the plane that contains also the plane of spikes. This subdivision is due to the fact that the plane of spikes has to move on the y axis while the circles have to remain fixed.

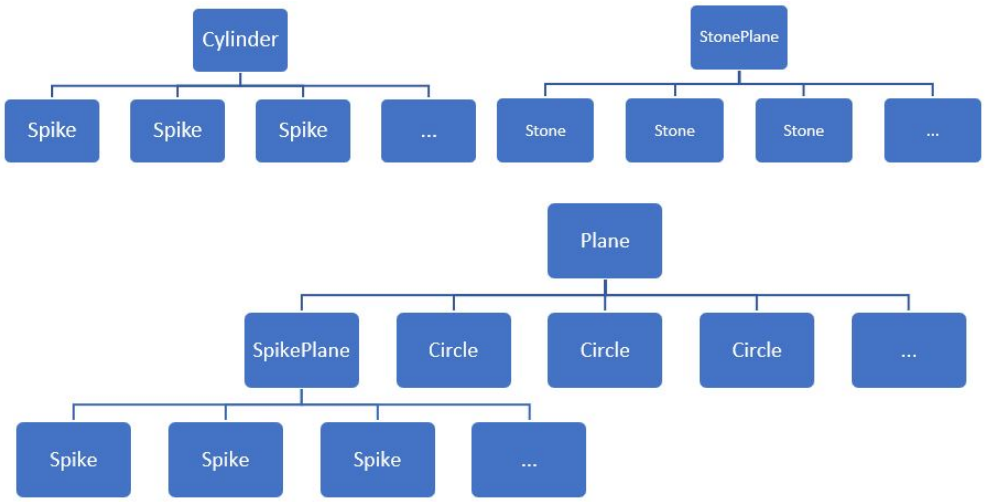
- '11': It represents 6 cubes that simulate pieces of the ceiling falling down on the floor. To create this obstacle I used the function "newStones(list[])". Each cube is a Mesh with "BoxBufferGeometry" as geometry and a "MeshBasicMaterial" (with a texture) as material. All these cubes are added as children of an Object3D that is returned from the function (this object will be added as a child of the main "scene") and pushed inside the input list of this function (the reason will be explained in the Collision detection section).

- '12': It corresponds to the Heart perk, that is a cube with a texture of a red heart on it. It is represented by a Mesh with a "BoxBufferGeometry" as geometry and a "MeshBasicMaterial" (with a texture) as material. The cube, added as a child of the main "scene", rests on an edge on the floor and rotates around the y axis.

- '13': It corresponds to the Invincibility perk. It is created in the same way of the previous element, the only difference is the texture that represents the Super Mario's star.

- '14': It defines the 2 ladders that the player has to hit to win the game. Each of them is a parallelepiped (Mesh with "BoxBufferGeometry" as geometry and "MeshBasicMaterial" (with a texture of a ladder) as material) and at the top of it there is a black plane that simulates the fact that it leads to an exit. This plane is represented by a Mesh with "PlaneBufferGeometry" as geometry and "MeshBasicMaterial" (with the color black) as material. The plane and the ladders are added as children of the main "scene" and the plane is rotated of 90 degrees to face the player (because it is put on the ceiling) and it is put slightly below the ceiling (to avoid the overlapping with the ceiling).

As I said in the light section, the main "scene" contains a white Ambient light and some points light. These last ones are placed every 4 elements of each row that is divisible for 4 and only if the corresponding element in the row is not a maze's wall. To highlight them I created a lamp object by using a Mesh with a "BoxBufferGeometry" as geometry and a "MeshBasicMaterial" (with a texture) as material. The lamp is put slightly below the ceiling and the point light is placed in the center of the lamp.



7 Implemented Interactions

I used three event listeners to manage the interactions with the user. I used one of them to manage the click on the screen on the "menuScene", in order to change the rendered scene from the "menuScene" to the main "scene" and the "overlayScene". I used the other two event listeners to manage the pressure of the keys on the keyboard. In detail:

-1: The "key down" event listener manages the following directional movements (namely the animations): moveForward, moveBackward, moveLeft, moveRight, Jump, Slide. For each run, I recorded the time instant in which the user presses a directional movement key for the first time (I used the flag variable "start" to understand when a new run starts) and I used it to count the total time of a single run. For each directional movement there is a flag that is set to true or false, depending on the fact that the corresponding key is pressed or not. I also used the "key down" event listener to:

- Change the camera view (by switching the value of the flag named "first") from the third person camera to the first person camera and vice-versa.

- Open the menu (by changing the flag named "menuFlag"). To open the menu the "menuScene" is rendered, instead of the main "scene" and the "overlayScene".

-2: The "key up" event listener stops the directional movements (moveForward/Backward/Left/Right) by setting to false the corresponding flags.

I added an extra event listener for the resizing of the window. I used this event listener to update the writings' size (such as the menu text, the revival text, the game over text and the winning text), the aspect or the dimension of the view of the cameras, the size of the renderer and the distance that the writings have to cover to disappear off the screen (see the animation section). The new dimensions of the texts are related to the smaller dimension (that can be the width or the high) of the resized window.

8 Animations

All the animations have been created by using a linear interpolation or by using a sin wave (I actually used the harmonic motion formula) in the "animate()" function. In detail, I created 4 main animations (the Walking animation, the Jump animation, the Slide animation, the Revival animation and the Text animation) and some other smaller animations (those for the obstacles). At the beginning of the "animate()" function I computed, through the variable "time", the current time of the frame that has to be animated. Then, I used this time instant to compute the actual position of the different objects in the environment.

In detail, the list of the animations:

1- Walking animation: I used the variable "walkClock" to compute the current time of each frame composing the walking animation. The variable "walkClock" is equal to the current instant of time of the actual frame (represented by the variable "time") minus the time instant in which the walking animation has started (represented by the variable "startWalk"). I updated the "startWalk" variable in the "key down" event listener only if its value was equal to "NaN" (the value "NaN" is assigned to "startWalk" when the player is in the resting position). In the "animation()" function, if the player is not jumping,sliding and it is not in the resting position, the value of "walkClock" is used to update 3 variables: "speedRotation" (equal to the value of the sine wave used for the animation), "upRotation" and "downRotation" (respectively equal to the values of the maximum angles of rotation around the x axis, that the upper part (Hips and Shoulders) and the lower part (Knees and Elbows) of the body player have to cover, multiplied by the "speedRotation". I created 2 auxiliary functions ("moveUpperParts(angle)" and "moveLowerParts(angle)") to rotate, around the x axis, the upper and lower part of the body of an angle equal to the parameter "angle" given as input of the functions. For example: if the right hip moves forward the right shoulder moves backwards and vice-versa for the left part. Then, the "upRotation" value can be set to positive for the "rightHip" and "lefShoulder" and to negative for the "leftHip" and "rightShoulder". The walking animation is performed when the player is moving around the environment. The direction of the player's movement on its z axis is represented by a value equal to the numerical value of the boolean "moveForward" minus the numerical value of the boolean "moveBackward". I stored this value in the z parameter of a "Vector3" called "direction". In this way, if the player is moving forward, the "direction.z" variable is equal to 1 and if the player is moving backwards, the "direction.z" variable is equal to -1 (the same method is used to compute the rotation, around the x axis, to the right or to the left). So, if the variables "moveForward" or moveBackward" are true, the new position of the player is translated by "-(direction.z/2)" units on its z axis (the minus can be explained by the fact that the camera, by default, looks the negative part of the z axis of the scene). If the player is performing a slide he cannot move backwards and the position of the player is translated back to him previous position. Finally, if the player is not jumping or sliding, the auxiliary functions "moveUpperParts" and "moveLowerParts" are called and the walking animation is performed.
The same mechanism is also used for the rotation to the right or to the left of the player. The only difference is that instead of a translation on its z axis, there is a rotation around the y axis equal to [actual angle of rotation - (degToRad(6)*direction.x)] (where degToRad() is an auxiliary function to convert the degrees in radians).
When the user stops pressing any key (all the flags for the movement's direction are false) and the player is not jumping or sliding, then the resting condition animation starts. In this animation all the body's parts continue their walking animation until each of them reaches the value 0 as angle of rotation around the x axis. When this happens, the flag variable "startWalk" is set to "NaN".
Since the rotation of the body's parts depends on a sine wave and on the time interval "walkClock", is quite impossible to have the perfect value of 0 radians for the rotation around the x axis. For this reason, I put two "if" conditions to verify when the upper part of the body and the lower part of the body reach a value of the rotation between -0.2 and 0.2 radians. When this is verified, I assign them the rotation value of 0 radians. Since the lower part of the body is slower to return to the resting position than the upper part then, I included these two "if" condition in another "if" condition. This new "if" condition is true only if the rotation value of the knee (that is the same value assigned to all the lower part of the body) is different from 0.

2- Jump animation: The jump animation starts when the user presses the "space" key. In the "key down" event listener the pressure of the "space" key changes the flag "jump" to

true. This animation doesn't use the "walkClock" variable but it uses the "jumpClock" variable, that is equal to the the current instant of time of the actual frame ("time") minus the time in which the jumping animation has started ("startJump"). The variable "startJump" is recorded in the "key down" event listener only when the "space" key is pressed and the player is not already jumping or sliding. The "jumpClock" time is used to update 2 values: "jumpVerticalSpeed" that is the value of the sine wave used for the vertical translation (on the y axis) of the player and "jumpSpeed" that is the value of the sine wave used for the rotation of the body's parts.
The vertical translation of the player is very easy to build: to assign the y position to the player in each frame of the jump animation, I used the variable "verticalPosition" that is equal to the Height of the jump multiplied by the "jumpVerticalSpeed". Regarding the rotation of each body's part, I divided the animation into two parts that I called "up" (or "not down") animation and "down" animation. The "up" animation part is when the player body's parts (upper and lower part of the body) go from any angle of rotation, depending by the walking animation or by the resting position, to the maximum rotation value of the angle for the jump animation. For example, if the jumping animation starts when the left shoulder has a rotation value equal to 60 degrees (due to the walking animation) then, the left shoulder will rotate from 60 degrees to its maximum rotation value in the jumping animation (set to 160 degrees). As the walking animation, even in the jumping animation, the upper part and the lower part of the body have the same maximum rotation value (70 degrees for the upper part and 90 degrees for the lower part). The only exception is for the left shoulder, whose maximum rotation value is 160 degrees instead of 70 degrees like the others parts composing the upper part. To compute the values of rotation from any position to the maximum rotation value in the "jumping" animation, I used the variables "upPartialRotation", "downPartialRotation" (respectively for the upper and lower part of the body) "upRotationJump", "downRotationJump" and "leftRotationJump" (this last one is the value for the left shoulder). In detail:

- "upPartialRotation" and "downPartialRotation" to compute the rotation values from 0 to a value that is equal to the difference between the maximum rotation value of the animation and the starting rotation value of the body's part. For example, if the right knee has an initial rotation value of 20 degrees and it has to reach its maximum rotation value that is 90 degrees, the "downPartialRotation" will compute all the values of the sine wave from 0 to 70.

- "upRotationJump", "downRotationJump" and "leftRotationJump" to compute the current values of the rotation. They are equal to the values of the corresponding sine wave plus their starting rotation value. For example, considering the knee, the starting rotation value is 20 degrees and the values of the sine wave are represented by the variable "downPartialRotation". Then, the final rotation will have an amplitude of 70 degrees starting at 20 and finishing at 90 degrees.

The condition to switch from the "up" animation to the "down" animation is the following: if the "down" flag variable is false and the value of the rotation of the upper part of the body is close to the maximum rotation value (value of rotation ¿= maximum rotation value - 1), then the variable "down" is set to true and the actual time instant of the current frame is recorded in the variable "startDownJump".
The "down" animation starts when the variable "down" is true and it consists in going from the maximum rotation value of the jumping animation to the resting position (namely all the rotations around the x axis of the player's parts have to be equal to 0).
As in the walking animation, even in the jumping animation, to return back to the resting position, the value of the rotation has to be close to 0. To make this possible, I put 4 "if" conditions (for the upper and lower part of the body, for the left shoulder and for the vertical position of the player).
In the first three "if" conditions, the rotation value has to be between -0.1 and 0.1 radians. When this condition is satisfied, the new value of rotation is set to 0 radians. On the con-

trary, for the vertical position, if its rotation value is lower than 0, then the position of the player on the y axis is set to 0. In the "if" condition of the "verticalPosition" (that is the slowest part to return to the value 0) I assigned the following values to exit from the jump animation: false to the flag variables "down" and "jump", the update of the "startWalk" with the current time of the frame and the update of the "walkClock" variable (this last 2 variables are needed if the player is walking during the jump animation. In this way the walking animation will immediately start after the ending of the jump).
Finally, the vertical position of the player, the upper and lower part of the body and the left Shoulder rotation values are updated.

3- Slide animation: The mechanism used to create this animation is the same used for the "jump" animation. The only differences are:

- I didn't use the auxiliary functions "moveUpperParts()" and "moveLowerParts()" to change the values of the body's parts, because I manually changed the rotation values of the parts involved in the animation.

- I put the values to exit from the sliding animation (the same of the jumping animation) in a single "if" condition, that represents the slower part of the body (again the "verticalposition"), instead of 4 "if" conditions.

4- Revival and Text animation: To move the texts I used the auxiliary function "moveText (timei, textObj)". This function uses the linear interpolation to move the input parameter "textObj" (which represents the text to move) on the y axis of the "overlayScene". The linear interpolation is created by using the following formula: [(arrival - start) * (timei-startTime)/(endTime-startTime) + start]. "arrival" is the ending position that I want to reach with the object, "start" is the starting position of the object, "startTime" is the time instant in which the animation has started, "endTime" is the time instant in which I want the animation to finish and "timei" is the actual time instant of the current frame. The starting time of this animation, stored in the variable "startRevival", corresponds to the time instant in which the player collides with an obstacle (see next section) and the ending time is equal to ["startRevival" + "bornDuration"]. The "bornDuration" variable is the duration of the revival animation which consists in adding and removing the player model from the main "scene". This animation creates a blinking effect that I used to simulate the re-spawn of the player. When the current time of the frame is greater than ["startRevival" + "bornDuration"] (it represents the duration of the revival animation), I remove the text from the "overlayScene", I set the "revivalFlag" flag to false and I add again the player model to the main "scene".

5- Obstacles' Animations: For all the obstacles, I individually changed their values of rotation and translation. I used different lists to store all the objects that have to move on the same axis or rotate around the same axis. For example, I used "xRotCylinders[]" to store all the cylinder objects that have to rotate around the x axis and then I updated their position by adding 5 degrees to their previous rotation value. Another example can be the usage of the values of a sine wave to animate the translation of the cylinders on the x,y or z axis or the linear interpolation to move the pieces of the ceiling that are falling down.

9 Collision detection

For each object in the maze, in order to compute the bounding box's bounds of the object, I used the bounds of a "Box3()" object, created with the function "setFromObject()". The procedure to extract the bounds of a "Box3()" object is very simple: first, I created the "Box3()" object from a maze's object with the function "setFromObject()" (by using the auxiliary function "createBounds()"), then I put the bounds' values of the "Box3()" object in a variable called "bounds" and finally I pushed the "bounds" object in a list that I used to verify if there was or not a collision with the player. In order to understand what kind

of object has collided with the player, I created different lists (one for each kind of object in the maze). For example, I created a list for the heart perks (called "heartPerk[]") or a list for the maze's walls (called "walls[]"). I used this approach for the fixed objects (such as the walls or the fixed cylinders), while for the moving objects I updated their bounding box's bounds. In order to do that, I created two auxiliary functions:

- The first function is called the "updateBoxes(boxes[],objects[])" and it updates the box object in position i of the list "boxes[]" with the object in position i of the list "objects[]".

- The second function is called the "updateHitboxes(boxes[],hitBoxes[])" function and it updates the values of bounds of the object in position i in the list "hitBoxes[]" with the box object in position i of the list "boxes[]".

The condition to check if there is a collision between two objects is: for each axis, the smallest value of the bounds of the first object has to be smaller or equal than the greatest value of the bounds of the second object and the greatest value of the bounds of the first object has to be greater or equal than the smallest value of the bounds of the second object.
The bounds of the player on the x and z axes are fixed and they correspond to the ones of the "Box3()" object created with the player in the resting position. On the contrary, the bounds on the y axis are updated in each frame by using those of the new position of the object. In detail:

- The lower bound on the y axis is the position of the "Object3D" called "player" (whose origin is allocated under the feet of the player's model).

- The upper bound on the y axis is equal to the sum of the lower bound on the y axis and the "y" parameter of the Vector3 called "yDim". This vector represents the size of the box created with the actual position of the player.

By using this shape of the player's bounding box, the user can move easily in the environment.
I created different functions to manage the collisions of the player with different objects.
1- "playerDetectCollision(list[])": This function takes as input a list of objects and verifies, for each frame, if the player has a collision with them. I used this function to check:

1.1 If there is a collision with the walls when the player is moving forward or backwards or when he is rotating to the right or to the left (if there is a collision the player is set back to its previous position).

1.2 If there is a collision with the obstacles. The collisions between the player and the obstacles are computed only if the "invFlag" variable is false. This variable is true when the player takes the invincible perk. When there is a collision, the "isGameOver()" function is called and it manages the loss of a player's life. In the "isGameOver()" function, the value "endGame" (which represents the current number of player's lives) decreases by 1. So, if the value of "endGame" is equal to 2 or 1, the writing of the revival animation ("you lost a life, be careful!") is added to the "overlayScene", the heart-shaped object (which corresponds to the i-th life of the player) in the "overlayScene" is removed and the "spawnPosition()" function is called. The "spawnPosition()" function sets the flags "jump", "slide", "down" and the directional flags (moveForwards,moveBackward,moveRight,moveLeft) to false, it sets all the rotations of the player's body to 0 and it puts the player's position in the spawning position. Otherwise, if the value of "endGame" is equal to 0 (it means that there is a game over), the writing of the game over ("GAME OVER, Your time is:" + total time of the run) is added to the "overlayScene", the functions "setStartingVariables()" and "spawnPosition()" are called, all the heart-shaped objects are added to the "overlayScene", the distance that the game over text has to cover is updated (the end game's writing is greater than the revival's writing, hence it needs to cover more distance before vanishing from the screen) and the "gameOver" flag is set to true. The function "setStartingVariables()" sets the "start" flag to true, the "endGame" variable to 3 and it

reinserts in the corresponding lists all the perks that have been taken by the player (see the function "invDetectCollision()" for more explanations).

1.3 If there is a collision with the ladders that lead to the exit of the maze. When there is this kind of collision, the flags "endFlag" and "menuFlag" are set to true, the writing of the end game ("You reach the exit! Your time is:"+ total time of the run + "Click on the screen to restart the game. Try to beat your time!") is added to the "overlayScene", the function "setStartingVariables()" and "spawnPosition" are called and all the heart-shaped objects are added back to the "overlayScene".

2- "invDetectCollision()": This function verifies if there is a collision between the player and an invincible perk. When the player hits an invincible perk, this function removes the bounds of this perk from the corresponding list containing the bounds of all the invincibility perks and it removes from their corresponding lists also the perk object and the box object. Then, it pushes all the three removed elements into three new lists (invRemovedBounds[], invRemovedObj[], invRemovedBoxes), later used in the "setStartingVariables()" function to put all the elements back in their corresponding lists. In this way, the collision between the player and this invincible perk isn't computed until the next run (after a game over or when the player reaches the exit). In the "animation" function, when the "invFlag" variable is set to true, the actual time instant of the current frame is recorded in the "startInv" variable, the color of the texture of the head and the torso of the player are set to red and the writing "invincibility for 10 seconds" is added on the "overlayScene". When the time of the current frame is greater than the value equal to "startInv + InvDuration", then the "invFlag" variable is set to false, the color of the texture returns to white and the writing representing the invincibility is removed from the "overlayScene".

3- "heartDetectCollision()": This function verifies if there is a collision between the player and the heartPerk.When the player hits an heart perk without having 3 lives, a heart-shaped object is added back to the player in the "overlayScene" and the variable "endGame" is increased by 1. As for the invincibility perk, even for the heart perk the bounds,the box object and the perk object are removed from their corresponding lists and added to 3 new lists (heartRemovedBounds[], heartRemovedObj[], heartRemovedBoxes).

The function "cameraDetectCollision(list[])", in the "animation()" function, detects when there is a collision between the third person camera and the walls of the maze (the walls' bounds are passed to the function by using the input list of the function itself). The bounds of the camera's bounding box are computed by using the camera's position. Hence, for each axis, the maximum bound is equal to the minimum bound. If there is a collision between the third person camera and a wall, the "firstPerson" flag is set to true and the main "scene" is rendered with the first person camera. Otherwise, the "firstPerson" flag is set to false and the main "scene" is rendered with the third person camera.

10 Other implemented features

1- "setSpawnPosition()": This function defines the starting position of the player, by creating a random number corresponding to one of the possible player's starting positions. This function also assigns the values of the player's starting position, for each axis, to the variables "xPos", "yPos" and "zPos". Finally, it puts the player's model in the starting position defined by the function itself.

2- Creation of the writings: For each writing I created a Mesh by using the "TextBufferGeometry" (with the corresponding string) as geometry and the "MeshPhongMaterial" (with the white color) as material. I aligned in the center the text and I added it to the corresponding scene. The menu and victory writing are rendered in the "menuScene" and the revival and game over writings are rendered in the "overlayScene". All the writings have the same font "$helvetiker_regular.typeface.json$" and text properties (font, size, height, curveSegments, bevelEnabled, and so on). There is no way to update the text of

a Mesh of this kind. The only thing to do is to create a new writing of this kind, delete the old one with the "removeTexts(s)" function and add to the corresponding "scene" this new writing.

3- "removeTexts(s)": This function removes from the "s" scene all the objects with the name equal to "text". In this way, I could remove the past writings on the "s" scene and I could add new ones. I had to create that function in order to avoid the fact that during the resizing of the window, lots of writings were added on the scene.