MAZE RUNNER

Discover the path, Triumph the maze!

Group Project Interactive Graphics

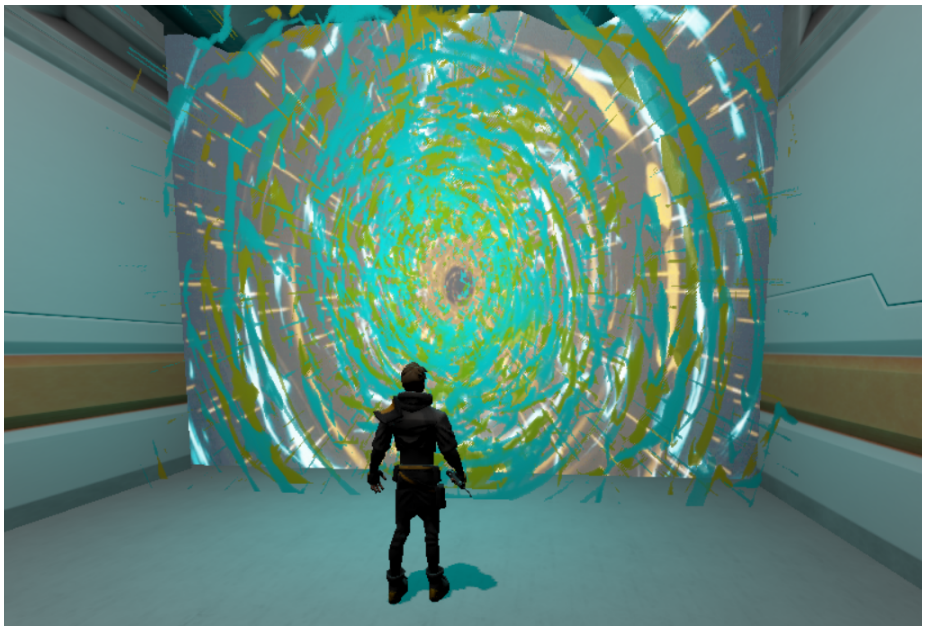
Avesani Davide 2060112
Pascucci Simone 1895398

16 July 2023

Contents

1 Introduction

Maze Runner is a video game in which players must traverse intricate mazes to find their way out.
The goal of the game is therefore to find and traverse the portal to escape the maze.



It offers 3 different labyrinths maps and 3 different characters selectables from the main menu.
The following paragraphs describe both the technical and playful aspects of the game.

2 Game play

The objective of the video game is to find a way out of the labyrinth.
Inside the maze there are various lights hanging from the ceiling and the player has a torch to help him illuminate the maze.
The user can Interact with both the torch and the lights by switching them on and off, by using the commands described in the following subsection.
In particular, only a few lights offer the possibility of interaction and are marked by a red button attached to the wall below them. When the player is in proximity of this red button, he can switch the respective light and off by using the appropriate button.

2.1 Commands

- Use the **W A S D** to move through the maze.
- Press and hold **SHIFT** to run.
- Press **SPACE** to jump.
- Press **ESC** to enter the pause menu.
- Press **F** to switch on and off the flashlight.
- Press **E** to interact with the objects.

2.2 Menu

This is the first page which will appear to the user. Here it is possible for him to choose between three different game modes and between three different characters. The page has been designed to have a little bit of interactivity and not to be only a static environment.
We decided to implement in our game other two menus other than the main one. In particular we will see a menu appearing on the screen when the user pauses his game and another one when you will win a game entering the final portal.
In the "Pause menu" as suggested by the name you will have a button which will let you resume your game from the exact point where you left and another one to completely restart the game by sending you back to the "Main menu". Other than this the timer on the screen will stop until you choose to resume your game. In the "Win screen" you will see the total time that you needed to find the end of the maze and then you will have a button to start a new run with your favourite settings.
The whole style aspect of the page has been designed using "Bootstrap" framework together with some basic CSS which can be found in the "index.css" file in the main directory of the project. But now let's see more in detail how the main menu has been designed.

Modalities:

The left side of the screen in the main menu has been reserved for the three game modes. As it's easy to understand to everyone of the three cards will correspond a different difficulty to find the end of the maze. In the maze generation part will be explained how the maze will be generated but we can anticipate that it will be considered like a matrix with a defined number of rows and columns. When you choose the game mode by pressing on its corresponding button you're basically setting the value of a variable that will set which maps has to be rendered. The number of rows and columns corresponding to every difficulty is defined as follows:

- **Easy:** 5 rows and 5 cols.
- **Normal:** 7 rows and 7 cols.
- **Hard:** 10 rows and 10 cols.

Characters:

The right side of the main menu has been designed to let the user choose between the three different character models. In particular this section is very interactive in a clever way. To every card has been assigned a different canvas element. Every one of these canvas have been used to render a different scene among three different ones using the Three.js library. Every scene has the same background image which is also the one that is used in the game itself. Then to everyone of these correspond a different loaded model. The loading of a model has been explained more clearly in the character paragraph. The most interesting part of every scene is the presence of the "Orbit controls". This controls provided by Three.js allows the users to interact with the character to see them from different points of view in order to properly choose the one that they want to use.

The game does not start until you click on the "Start game" button at the center of the screen. This will generate the chosen maze and will load the chosen character, but if you click on it before you choose both the character and the maze an alert will pop-up on the screen remembering you to choose the game mode, the character or both of them.

3 Maze

The player needs to navigate through a maze to win the game.
To add diversity and interaction to the game play, three different mazes of varying complexity have been added.
Note that the generation of the 3D maze is automatic and correctly works with every set of points that defines a 2D maze given in this format:
$[x_1, y_1, x_2, y_2]$.
In the next paragraph we are going to discuss how these maps has been generated and visualized.

3.1 Maze generation

To generate the maze we relied on this GitHub repository that allows to generate and solve random 2D mazes of different sizes and complexity.
This generator uses an algorithm to generate a series of points that define the lines of the labyrinth. By connecting these points with their respective lines, the 2D maze is drawn on the screen.
By making a few changes to the code, we were able to obtain the array containing the set of points used to generate the maze.
We then used these points to create the 3D maze, as described in the following section.

3.2 Adapting to 3D environment

The objective of this paragraph is to explain how we transformed the set of points defining the 2D maze in a 3D model.
The goal is therefore to translate a set of 2D coordinates in the form $[x_1, y_1, x_2, y_2]$ to the three.js 3D object THREE.BoxGeometry which takes as parameters (width, height, depth). This object will form the walls of the labyrinth.
The height and depth are standard for each wall and we defined it according to our needing.
The width is instead calculated as the distance between the two points.
In this way, the generated wall has the correct dimensions, but is generated in the origin and placed perpendicular to the X axis.
We need therefore to proceed by translating it to the right coordinates and eventually rotate it by $\frac{\pi}{2}$ if the line connecting the two points is horizontal.
Once doing that we obtain the structure of the final maze.
To give the walls the appearance of a real wall, we proceeded to generate the geometry by applying a texture, a bump map, an ambient occlusion map and a displacement map provided by this website.
Finally we generated the ceiling and the floor always by using a THREE.BoxGeometry of the required dimensions and generating also the geometry with the same technique as the maze walls.

3.3 Other Maze elements

In the initial maze structure then we had to add other two walls. One at the start of the maze which we decided to leave not visible in order to let the user to see the background, simulating a maze that is floating in the space. The other wall has been added at the end of the labyrinth. We decided to attach a portal texture to it. We wanted to recreate the effect of a GIF texture on the portal so we had to create three different meshes, one static on the background and two animated which only contain the frame of the portal and rotate to recreate something like a "Black hole" effect. Other than the maze structure, the ceiling and the floor, there are other two elements that composes the maze which are ceiling lights models and the red buttons which are used to switch on and off the lights.
in particular, the ceiling lights model uses this model model. They are randomly placed around the corridors of the maze.
We used the same website as the ceiling light to download the button model and we randomly placed it on the walls in correspondence of a light model.

3.4 Collisions

To handle the collisions we used a Three.js method called intersectsBox. It can be called from a Three.js Box object and takes as parameter a second Three.js Box object. It returns true if the two boxes are colliding, false if not.
The idea is therefore to check if the character box collides with any of the wall boxes.
The first step is therefore to create a Box for each wall and one for the character. In order for intersectsBox to function correctly, these boxes must be updated each and every rendering frame with the current value of matrixWorld.
By saving the direction in which the character was moving before colliding with a wall, if the collision is detected we prevent the object from proceeding in that direction. In this way we prevent the character from passing through the walls of the labyrinth.

A feature of the game is allowing the player to interact with the red buttons placed on the Maze walls. To be able to interact with them, the player must be within a certain range near to the buttons. To apply this game mechanic, we used a technique similar to the one used for wall boxes collisions: we defined a box around the red button as large as the proximity area in which the player must remain in order to interact with the object. If the model box collides with one of the red button boxes, it will be able to interact with them and thus turn on/off the respective light.

4 Character

The code regarding the character can be found in two different javascript files under the components directory. In the "character.js" file can be found everything regarding the movement, the camera updates and the collision managing, while in "animation.js" can be seen how the models are loaded and how they are animated frame after frame. The different character models can be accessed by clicking on this link: Character models. As can be seen these models are already "Rigged", this means that they already present a hierarchy in order to properly animate them using classical transformations instead of using pre-defined animations. We had the opportunity to choose to download the models in two different format which were FBX and GLB. We choose to use the glb format because it is more optimized for the usage along with the Three.js library. The models are loaded using the "GLTFLoader" class provided by the library, in particular the path of each character model is initially stored into an array and then only the chosen one will be loaded. To the character then will correspond a "State" of the following:

- **Idle**
- **Walking**
- **Running**
- **Jumping**

To every one of these status will correspond a different animation. This aspect will be explained deeply in the next paragraph.
Initially the character will be placed at the beginning of the maze and will be in the "Idle" state, the it will be movable using the W A S D keys which will change its position and its orientation. Once that the model of the character will be completely loaded we will also load a model for the flashlight that will be attached to the right hand of the character by defining it as a child of the right hand of this one, so they can be moved together.
Other than the loading of the models and the computing of its movements and its animations another important aspect of the character was the computation of a bounding box around the model with the purpose of correctly handle collisions with the walls, the buttons and the end portal. The box has been created to be slightly larger than the model and has been linked with it using a Three.Group, in this way when the position of the model changes, also the one of the bounding box will change. To achieve the same result we could have also used a library to implement physics such as "Cannon-es".

4.1 Character animation

As anticipated before, in the "animation.js" file we can find everything regarding the animation of the character. We defined four different statuses to which corresponds three different animations. In particular we have an animation

for idle, waling and running, and the running animation will also be used for the jumping state. In order to implement each animation we got the desired body part by the "model.getObjectByName(¡Bone¿)" function. Then we apply to this body part the desired transformations (Translating and rotating) using mathematical functions like sine and cosine combined with the "Elapsed time" in order to have realistic animations. Intuitively we wanted our character to slightly move the head and the body when in idle, to move also the legs and the arms when walking and running, along different axes and with different speeds relative to each state.

5 Camera

In this section we are going to see which type of camera we are going to use and how we intended it to work and how we implemented this aspect in our project. More in depth we used a classical Three.js perspective camera and we wanted it to work in this way: We have the camera constantly looking the back of the character with a slight offset.
We are also free to move the character in every direction without having the constraint to move it along a specific axis. In order to achieve this result we had to make use of quaternions, which are useful to calculate the current character orientation along each axis. As can be seen in the character class, in particular in the "movement" method, it is also responsible for the camera position and "lookAt" updates. In particular we choose empirically the starting position and look at vectors to which we add the current character orientation using "character.quaternion" property. Then we set this values called "Ideal offset" and "Ideal look at" each frame as the camera position and lookAt. In this way we can achieve the desired result explained before.

6 Lights

We used several lights to lighting the scene and add more depth to the scene.
In the following section we will describe how we used the various lights types.

6.1 Ambient Light

A very weak ambient light is added to the scene.
It is used to simulate the fact that a small portion of lights enter the labyrinth from the starting and the ending point. It is however extremely weak since we do not expect that these light sources can lightening much all the corridors of the maze.

6.2 Spot Light

We used a Spot Light to simulate the light emitted by the torch.
We set all configurable parameters (colour, intensity, distance, angle, penumbra and decay) in the best way for the purposes of this video game.
By making the Spot Light son of the Torch we obtain the effect that the spot light follows the position of the character.
In order for the cone of light to point in the direction in which the character is looking, we set the position of the light target as the player's point of view.

6.3 Point light

The point lights are used to simulate the light emitted by the ceiling light model. They are therefore equal in number to the generated ceiling lights and positioned above the respective light. In this way we obtain the effect that the light models emits a light.
We also used two other point lights to simulate a dim cyan light entering from the initial and the ending point of the maze.

6.4 Shadows

NOTE: This section only refers to the game easy map. This is because it is computationally too heavy to load all the shadows generated by all the lights and walls presented in the normal and difficult maps.
We set that the character projects the shadows generated by the point lights on the ceiling on both the maze walls and the floor.
The maze walls not only receives the character shadows but they also generates a shadows projected also in this case to both the floor and the other walls.