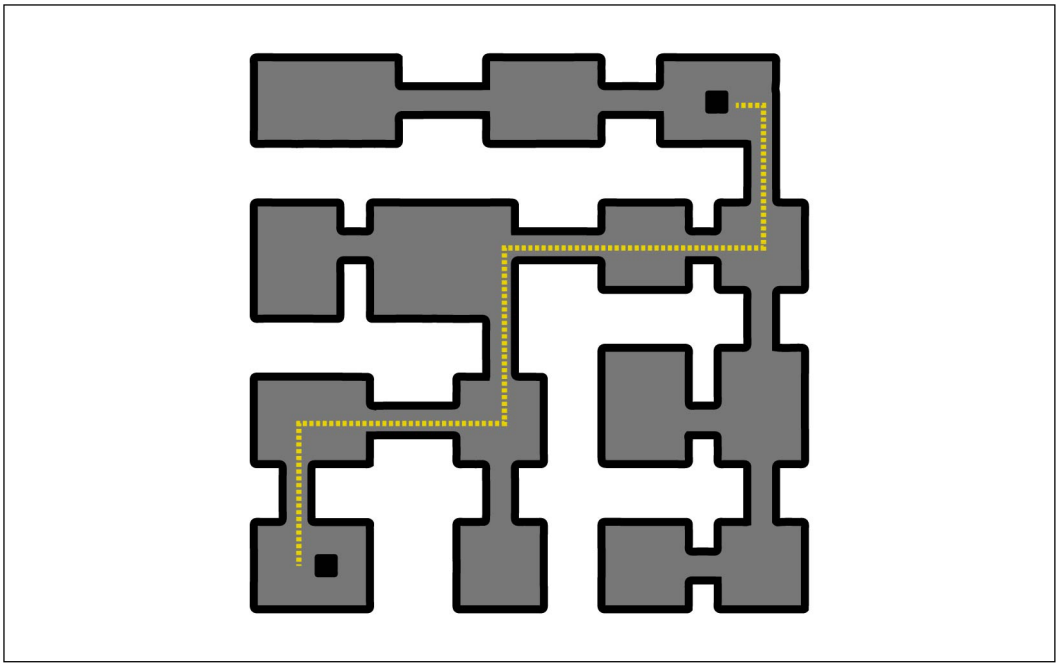
Escola Politècnica Superior

Universitat de Girona

Project ADD: Adaptative Difficulty Dungeons

Trabajo final de grado

Grado en Diseño y Desarrollo de Videojuegos



Document: Report

Author: Elio Matías Valenzuela Muñoz

Director: Gustavo Ariel Patow and Nicolas Barriga Richards

Department: Informàtica, Matemàtica Aplicada i Estadística

Area: Llenguatges i Sistemes Informàtics (LSI)

Call: September/2020

Contents

Chapter 1

Introduction

For a lot of people, playing video games is something that allows them to have many hours of entertainment. In general, games test the skills, instincts and strategies of the players to make them fall into a state of pure concentration or complete absorption called "Flow" [10].

When players want to start a game, they choose among the different difficulty levels the one that better fits their skills to start playing. However, those difficulties are created by the game designers, indicating that, in order to play in that difficulty, the player must develop a set of skills that fit that difficulty. That minimum skill level has been already set by the designer.

These two conceptions of difficulty and skill are different for each player and designer. Players may think that they possess the skills to overcome all the challenges and enjoy the game in the difficulty that they have already chosen. But, what happen when a player chooses the max difficulty and is not as hard as he thought it will be, or if the player want to start the game for the first time and the easy difficulty is not that easy? Or if the player develop the skills faster or slower and get bored of winning or losing too much? Players who are struggling with a difficulty will no longer want to play that game anymore, they get bored and that is the last thing, that as a game developer we want our players to feel. In Figure 1.1 we can see the ideal experience that the player should have during a complete game session and how the game objectives should be balanced with challenges according to the player skills.

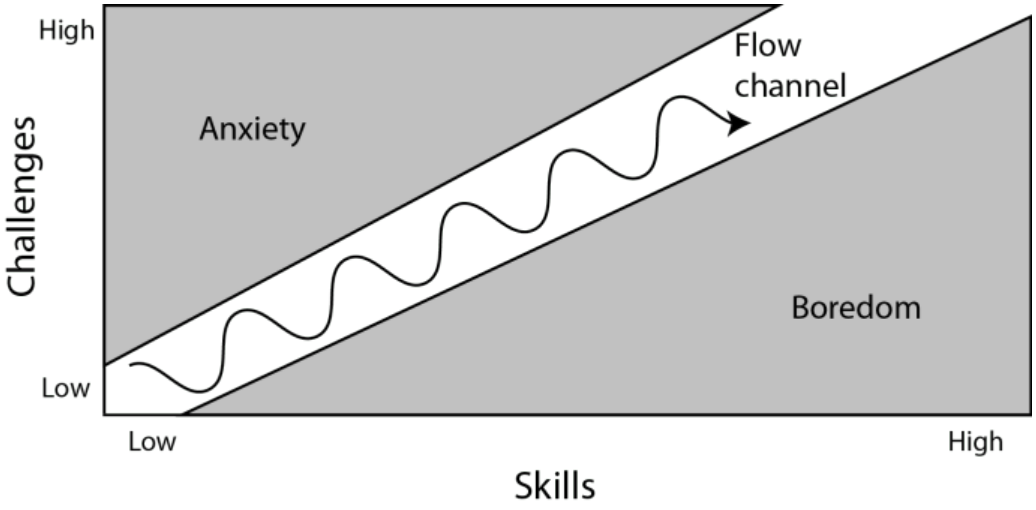


Figure 1.1: Flow Channel graphic from The Art of Game Design: A book of lenses [5]

When the presence of challenge in the game is low, players get bored and when repeatedly confronted with defeat, they get frustrated and possibly lose the interest in the game. In this work we will focus our efforts on analyzing the player "skills" and quantify the difficulty of the level structure. We will find a way of generating content with a level of difficulty that fits the player's skill level.

1.1 Personal Motivations

Having to go back and work in the level design, the mechanics, the enemies or even figuring out why a player is struggling to complete a specific part of the game could make the process of developing a game a really tedious work. The design of how we make a game more difficult to the player could make that other players feel left out of the experience that we were creating and miss the opportunity of enjoying a new game.

I have always thought that the difficulty of games is not adequate for everyone. Some people would enjoy playing on easy, medium or hard, but what about the rest of the people? There are people who like to play on a difficulty more challenging than the hardest, or easier and more calm than the easy mode to enjoy the story or learn to play the game on their own rhythm.

That is why we want to develop a tool that could help to create a more personal content for every player. With a more personal difficulty scaling, a lot of players could start to play and integrate in the game mechanics developing the skills necessary to overcome all the challenges and difficulties on their own pace.

1.2 Scientific Motivations

Capcom's Resident Evil 4 [1] changes the difficulty of the game dynamically to match the player skills. The game makes the enemy attacks slower or faster, resist more or less bullets, giving more or less ammo or even reducing the enemy numbers in the levels. These are some of the things the game does to make the game easier or harder. Mario Kart 8 [2] makes changes to the AI if the player is in the last position. Also, the AI will not take the shortcuts and will not take the curves through the inside if the player is close to them. In Valve's Left for Dead 2 [3], the map changes the spawning points were the enemies appear and opens or closes sections to make experience through the level unique in every playthrough.

Other games have no difficulty levels or, at least, the player has not the option to change the difficulty of the game. Games like Dark Souls [4] or Mario Odyssey [5], both of different game genres but without a game difficulty setting. We know that it is the way the game was intended to be experienced, but with this complexity we could have many other players, who, without a difficultly level in which they feel comfortable, most of the time will lose, and get frustrated or even get bored at some point if the game is not challenging enough.

There were several games that do not have a difficulty level and create a variety of content

[1] Fourth entry in the Resident evil series who introduce the Dynamic Difficulty into the saga.
[2] Mario's Racing Franchise
[3] Multi-Platform Game and Second entry of the Series
[4] Action-RPG game
[5] Last entry of Mario's Games

like Dead cells [6] and Spelunky [7], which create different game levels and areas with different enemies and rewards. But those games, even without a difficulty level, will be difficult from the start with the player learning about the game mechanics. The content is not normalized, so it only depends on luck if the player stops by a really hard enemy or a really useful reward.

In this work, we want start to create a tool that could make video games easier or harder with the creation of personalized content with Dynamic Difficulty Adjustment (DDA) [10] technics, that consist in automatically modify aspects of the game in order to adjust the difficulty of the challenges and objectives that the player complete along the game. We are going to use them to build an entire level from a very low scale to the whole game experience, creating the complete level structure, the enemies and controlling their stats and behavior, challenges that will be difficult but they could overcome it because the will have a whole system that supports and analyze their actions without them knowing about it.

1.3 Purposes and Objectives

The purpose and main objective of this project is to make a first approach to the creation of a tool that uses DDA techniques to generate personalized content. Using path finding algorithms such as A* and Breadth-First Search (BFS), we plan to find relations that help us interpret the player's skills and the procedurally generated level difficulty.

[6] Rogue-lite metroidvania inspired game
[7] Platform game who create randomized levels

Chapter 2

Feasibility study

This chapter will describe the Technical requirements and human resources necessaries along with the tasks and the total cost of production.

2.1 Technical resources

The project will be developed on a single device with the following specifications:

Brand	Acer Predator Helios 300 (2017)
OS	PC, Windows 10 64 bits
RAM	16 GB
Processor	Intel Core™ i7
Graphics card	Nvidia GeForce RTX™ 1060
Storage	256 GB SSD + 1TB HDR

Table 2.1: Main device specifications

2.2 Human resources

For the proper development of the project, the followings profiles will be required, each one of the listed profiles is with its associate costs per hour.

- Programmer: 10 €/h.
- Analyst: 12 €/h.

Task	**In Charge**	**Hours**	**Total Cost**
Investigation: DDA Systems	Analyst	112	1334 €
Investigation: PGC Systems	Analyst	112	1334 €
Task divition	Analyst	56	672 €
Binary Tree Partition implementation	Programmer	112	1120 €
Procedural Dungeon implementation	Programmer	56	560 €
A* algorithm implementation	Programmer	112	1120 €
BFS algorithm implementation	Programmer	112	1120 €
Game loop	Programmer	112	1120 €

Continued on next page

Table 2.2 – *Continued from previous page*

Task	**In Charge**	**Hours**	**Total Cost**
Player's Data gahtering system	Programmer	112	1120 €
Testing / Gather Data	Analyst	240	2880 €
Data Analysis	Analyst	168	2016 €
Documentation	Analyst	160	1920 €
TOTAL			**16316 €**

Table 2.2: Tasks

The programmer will be in charge of every task that requires implementing algorithms and programming systems, helped by the analyst whose main task will be investigate about other projects related with dynamic difficulty and procedural content generation, and to analyze the data gathered and create a way to quantify the player's skills and the difficulty of the levels

2.3 Technical requirements

The followings software will be needed to develop the project and make the analysis of the information:

Item	**Price**
Developer PC	850 €
Unity3D (Personal)	0 €
Visual Studio Code	0 €
Excel (Student)	0 €
Phyton 3 (Graphs)	0 €
Github	0 €
Google Drive	0 €
Trello	0 €
TOTAL	850 €

Table 2.3: Hardware and Software

The majority of the software to be used will be completely free or a demo will be enough to carry on the development of the project. The device is able to run all the software mentioned in this section.

2.4 Total Cost

In total the cost of the project will be 17166 €. In the hypothetical case of hiring employees, the tasks table will be a guide of a more adjusted remuneration.

Chapter 3

Methodology

We work with a incremental methodology, separating the project into different sections, tasks and subtasks and advancing in stages predefined by ourselves. We can see, in the following flow chart 3.1, the methodology applied for this project.

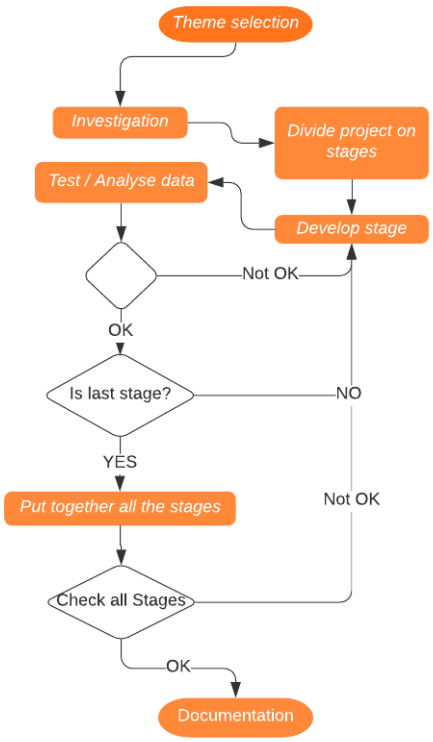


Figure 3.1: Flow Chart

On every task we make an analysis of what we need to proceed, and then we design the structure of the corresponding script, complete a feature or a behavior of a object. The design process was always fast because of the nature of the algorithms that already exist and the simplicity of their structures (BTP, A*, BFS, etc.). The next step was coding and implementing the design framework into the project. Once the behavior was done, we proceed to test if there are any problems or bugs. If not, we mark the task as completed an continue with the next one [1].

Chapter 4

Project planning

In this chapter we are going to show the final schedule to complete our work and task that we present in earlier chapters.

4.1 Final Planning

Due to the COVID-19 pandemic we had to travel back to Chile and, because of that situation, all our work was delayed. We had to re-evaluate do the presentation of this project in June, and we decide to finish the investigation and present it in September, hoping that this situation will be over, or at least improved by that time.

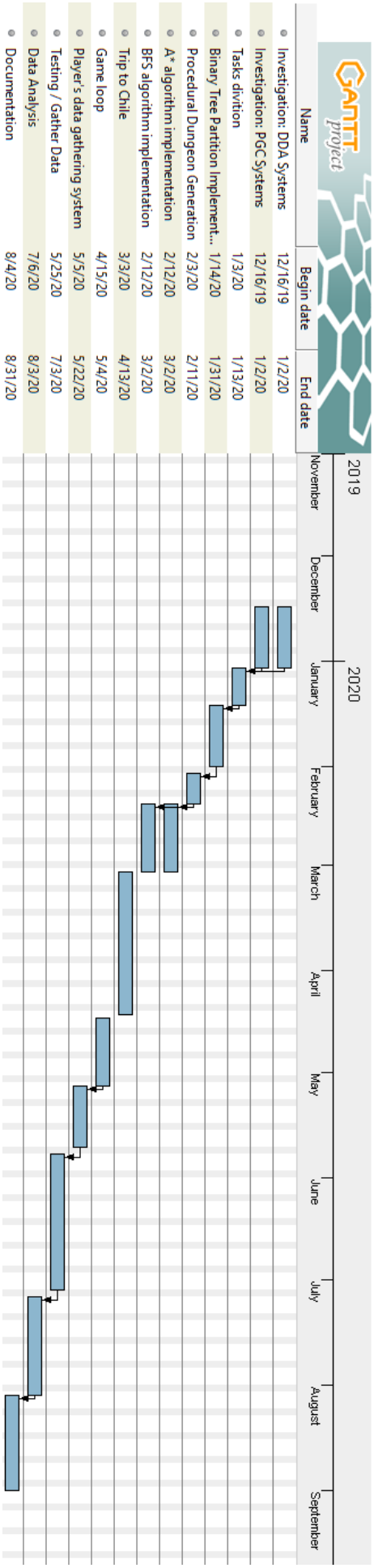


Figure 4.1: Final Planning

Chapter 5

Framework

This chapters present the background that is needed in order to understand the project. We are going to present an overview of the multiple algorithms, software and data structure that we use in this project.

5.1 Procedural Content Generation Algorithms

In the study of procedural content generation, there are different methods for content creation. We wanted to create levels from scratch and get the maximum content variation possible with different room structures and connections between them. There is a variety of options to do it like the constructionist approach, who take pre-created pieces of building blocks to create complete rooms and levels. Another type of content creation approach is constraint-driven, in which we set a number of variables (constraints) and we generate the content based on those variables. In this case, we set the size of the maze and the minimal size of the rooms [7]. We finally choose to use a constructionist approach because we wanted to create dungeon like levels like the one showed in Figure 5.1 [1], so in further develop, we can place more and more content into the rooms of the level.

Figure 5.1: Dungeon like maze example

There is a vast knowledge about methods for maze generation, one of the simplest being the Depth-First Search (DPS) that creates a series of passages with dead ends randomly until

[1] Darkest Dungeon level structure

there is no node left to visit. Other possible method was Prim's Algorithm 5.2, which makes interconnections between nodes and creates passages between them. DPS is more like a digger; it randomly carves the paths of the maze and, when it reaches a dead end, it traces back to dig another tunnel until there is no space left to dig. Those two approaches result in a maze much like the one in Figure 5.2 [8].

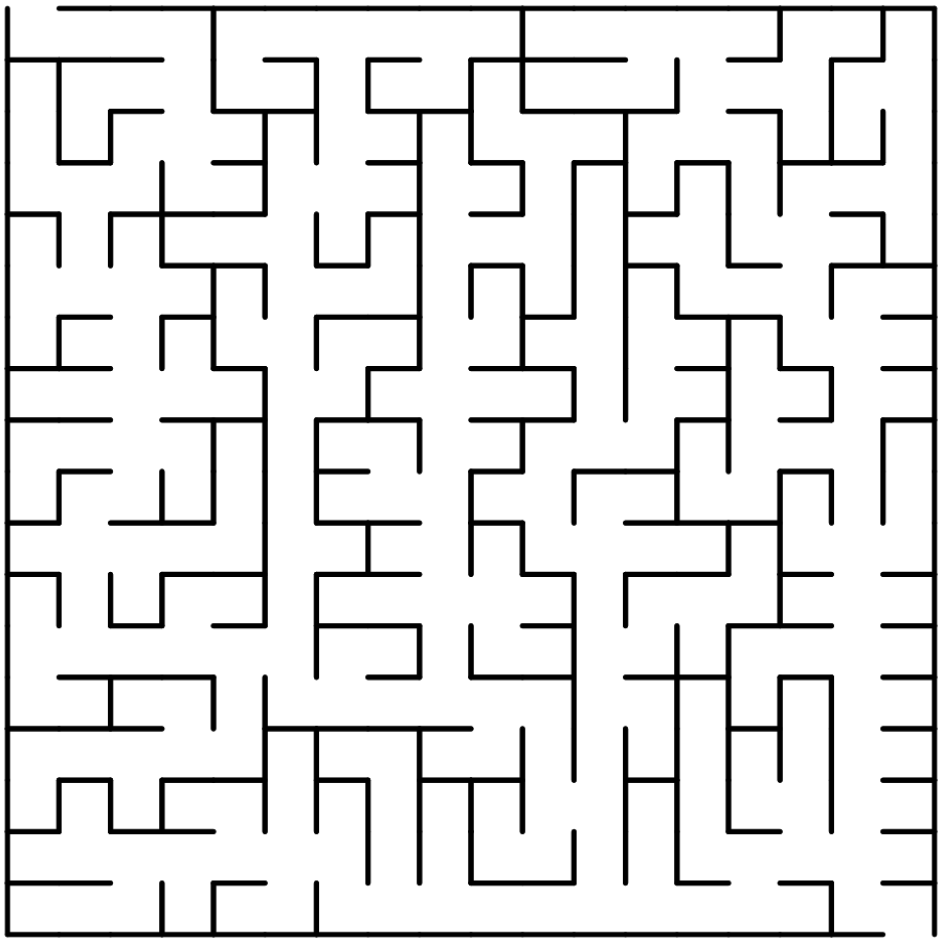


Figure 5.2: Result of Prim's maze generation algorithm

We analyzed different approaches for maze generation that gives us different results. However, since we want to create dungeons-like mazes, we finally used Binary Tree Partition (BTP) to create rooms and connect them to give the mazes a dungeon-like distribution.

5.1.1 Binary Tree Partition (BTP)

BTP is implemented to recursively subdivide a space in two partitions, see Figure 5.3. Those partitions, in the context of dungeon generation, are subdivided until a minimum room size is reached. Once the partition process is finished, we proceed to connect the rooms.

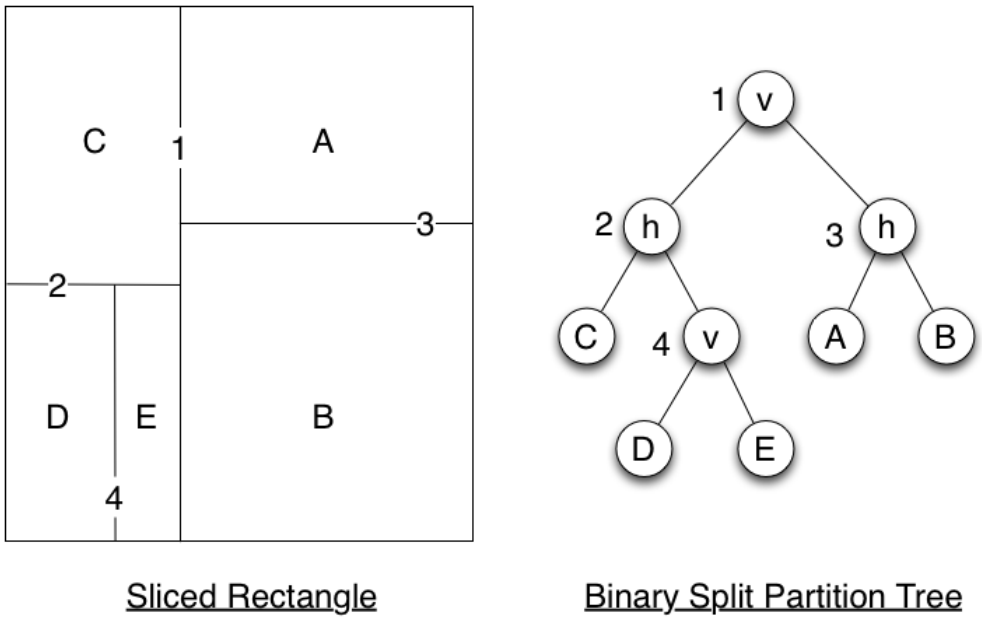


Figure 5.3: Binary Tree Partition

In Figure 5.3 we can see how it creates a subdivided maze. Then, we connect those rooms with corridors and create the final dungeon. We choose two rooms to place the entrance and the exit of the maze. We make the distance between them to have at least half of the maze size as minimum. Next, we show a pseudo code [6] that helps to understand the implementation.

Algorithm 1 Binary Tree Partition pseudocode

```
1:  procedure BTP
2:      start with the entire dungeon area
3:      divide the area along a horizontal or vertical line
4:      select one of the two new partition cells
5:      if this cell is bigger than the minimal acceptable size then
6:          Go to step 2
7:      end if
8:      select the other partition cell, and go to step 4
9:      for for every partition cell do
10:         create a room within the cell by
11:         randomly choosing two points within its boundaries
12:     end for
13:     starting from the lowest layers, draw corridors to connect
14:     rooms corresponding to children of the same parent in the BSP tree
15:     repeat 9 until the children of the root node are connected
16: end procedure
```

We see how, in the pseudo code, we start with an entire map filled with walls. This is our root node. Then, in line 3, we split the entire maze in two and we take one of them and keep splitting until we reach our desired size. Then when, we stop splitting, in line 9 to 11, we start from the last partition and start to create rooms within them, so every leaf node of the graph will be a room. Finally, in the line 13, we see how we proceed to connect the rooms until we reach the root node of the algorithm. The BTP method relatively simple to implement and following this pseudo code will grant you a very solid code.

5.2 Path finding Algorithms

In this section we will discus the pathfinding algorithms used in this work, we use A* and Breadth-First Search (BFS) because of their similarities and their data will be useful to complete our work objective.

5.2.1 A*

A* [2] is a very straightforward algorithm for path finding. Its main feature is its heuristic estimation of the distance to the goal. This allows the algorithm to only cover the necessary nodes and, every time it open a node, it calculates how far it is from the exit from it current position. A* uses three important values that are necessary to truly implement the algorithm, the 'g' value that refers to the total distance from the initial point to the current one, the 'h' value that is the heuristic value of the algorithm, which in this case is Manhattan distance, and the 'f' value that is the sum of both the h and the g values. Figure 5.4 [4] illustrates the result of using A*.

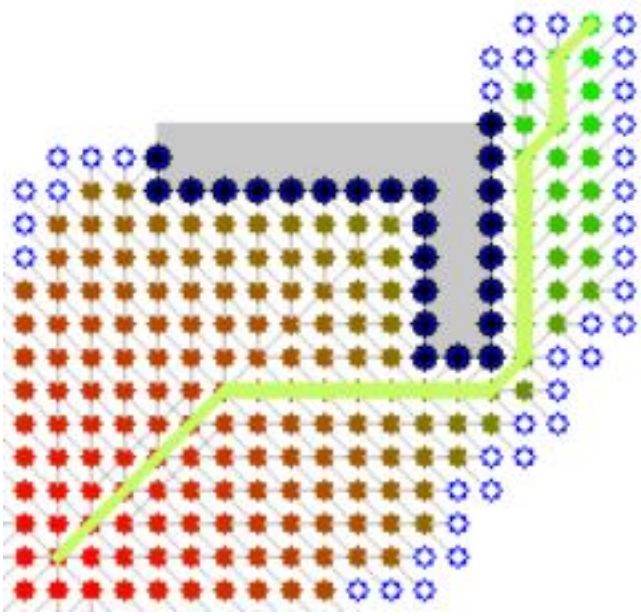


Figure 5.4: A*, the green line is the optimal path, the colored nodes are the nodes in the closed list and the blue hollow ones are the nodes in the open list

We use this algorithm to create the entrance and exit of the generated maze. From the algorithm, we obtain the data of the Open and Closed nodes, and the optimum path to reach the exit. Next, we present the pseudo code in Algorithm 2 to better understand the method.

[2]Pronounced A-star

Algorithm 2 A* pseudocode [4]

```
1: procedure ASTAR
2:     let the openList equal empty list of nodes
3:     let the closedList equal empty list of nodes
4:     put the startNode on the openList
5:     while the openList is not empty do
6:         let the currentNode equal the node with the least f value on the open list
7:         eliminate the current node from the openList
8:         Add the currentNode to the closedList
9:
10:        if currentNode is the goal then
11:            Backtrack to get path
12:        end if
13:        let the children of the currentNode equal the adjacent nodes
14:        for each child in the children do
15:            if child is in the closedList then
16:                continue to beginning of for loop
17:            end if
18:            child.g = currentNode.g + distance between child and current
19:            child.h = distance from child to end
20:            child.f = child.g + child.h
21:            if child.position is in the openList's nodes positions then
22:                if the child.g is higher than the openList node's g then
23:                    continue to beginning of for loop
24:                else
25:                    update the child on the open list
26:                end if
27:            else
28:                add the child to the openList
29:            end if
30:        end for
31:    end while
32: end procedure
```

We start with the initial node, in lines 6 to 13, we calculate its f and we search for all his neighbors. Then, in lines 14 to 17, for each one of them, we verify if is in the closed list, and if it does, we skip it. If not, we calculate its 'g' and its 'h' values in lines 18 and 19 and then its 'f' value in line 20. Finally, if the node is in the open list, we keep it in the loop. If not, we update its values.

5.2.2 Breadth-First Search (BFS)

Breadth-First Search is used to search for connected components of graphs. It uses a different strategy than Depth-First Search. BFS starts at any node, but we will start at the entrance and explores all the neighbor nodes before moving to the next level neighbors of each node. See Figure 5.5.

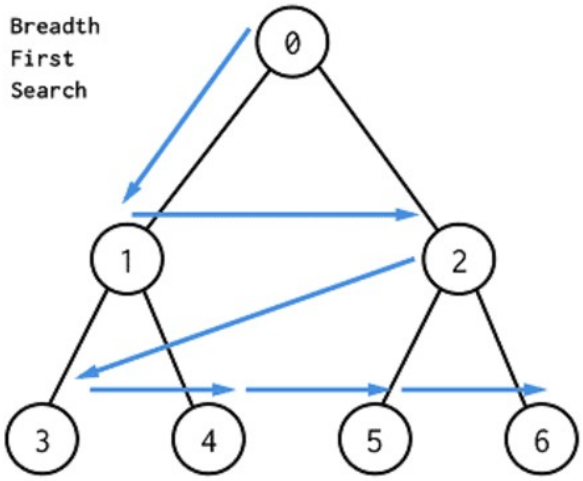


Figure 5.5: BFS graph representation

To implement the algorithm, we use the same algorithm as in A*, but without the heuristic. This results in an algorithm that searches for the exit but without knowing how far it is or if it is getting any closer. This makes the data of the Open and Closed node list different, but in both cases the optimal path will be the same. In the case of the maze path finding, both algorithms always find an optimal path. The difference is in the expanded nodes; while A* uses an efficient heuristic, like in our case Manhattan distance, the nodes will always be less than in the BFS expanded nodes [9]. In conclusion, both will find the exit with the same optimal path, but with different expanded nodes, with BFS always being the one with most opened nodes.

Algorithm 3 BFS pseudocode

```
1:  procedure BFS(G, root)
2:      let Q be a queue
3:      label root as discovered
4:      Q.enqueue(root)
5:
6:      while Q is not empty do
7:          v := Q.dequeue()
8:          if v is the goal then
9:              return v
10:         end if
11:         for all edges from v to w in G.adjacentEdges(v) do
12:             if w is not labeled as discovered then
13:                 label w as discovered
14:                 w.parent := v
15:                 Q.enqueue(w)
16:             end if
17:         end for
18:     end while
19: end procedure
```

The BFS pseudo code is very similar to the A* in the use of a list or a queue to analyze all the nodes of the maze, but in BFS we do not use values to find the closest node to the exit or our current position cost. We start with the initial or root node. Then, we start the loop

in line 6, while our list is not empty, we search for the exit. We expand our search into the adjacent nodes of the current one. In line 15, we add it to he list if is not already in it, and we keep searching for the exit until we find it.

5.3 Unity3D

Unity3D is a game engine able to create video games and animations. It supports 2D and 3D games. We use Unity3D basic features to create a simple game able to gather all the information we need from the player. Unity allow us to create builds of our project on different platforms, including the web. However, we are going make a Windows build because of the difficulties to create a web build are that we should edit a lot the sections of our data management module script to make it correctly works in the web platform.

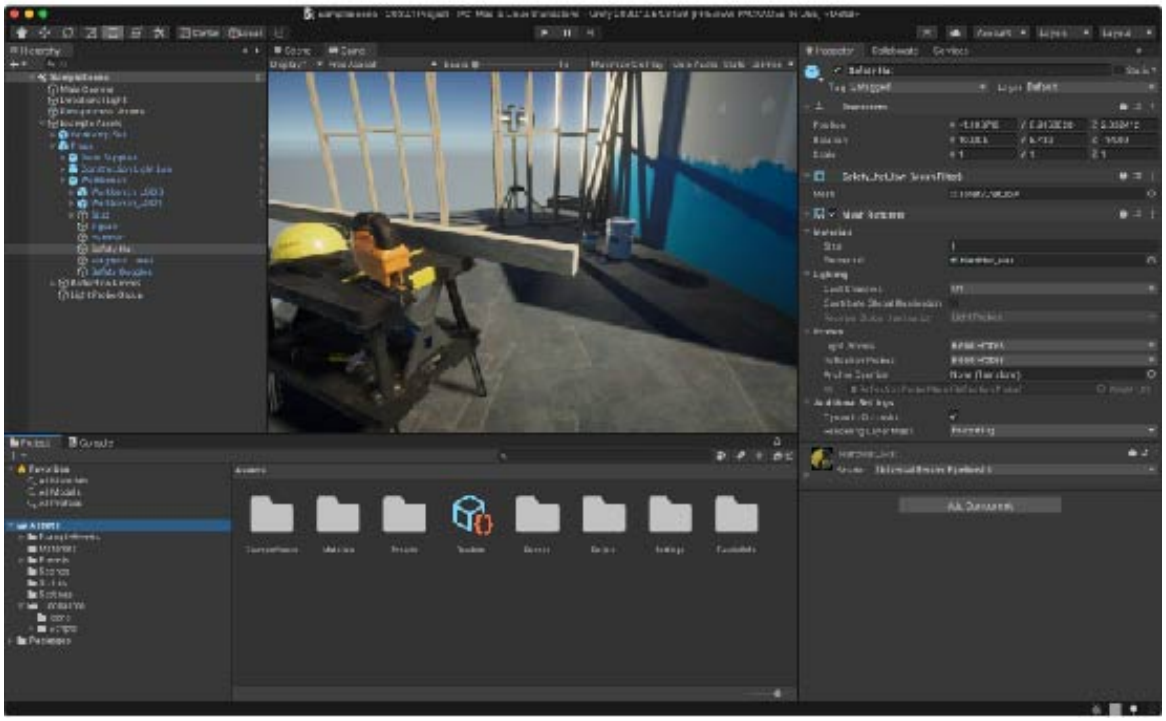


Figure 5.6: Unity3D and its editor

Chapter 6

System requirements

In this chapter we show the functional and nonfunctional requirements of the system.

6.1 Functional Requirements

In the game, the user should be able to:

- Walk around the map
- See around the map
- Start and finish (reach the exit of) a level
- Send the data of the player's playthrough
- Verify the player score and data
- Change the sound and display options of the game

Our main goal is to analyze the player times and movements values, so walk and move the camera around are the basic and most important functionalities of the game, along with send the data to our database.

6.2 Nonfunctional Requirements

This requirements refer to the requirements that needs the player to have in order to play the game.

6.2.1 Hardware

To play the game those are the minimum hardware requirements:

OS	PC, Windows
RAM	4 GB
Processor	Intel Core™ i3 or higher
Graphics card	Nvidia GeForce GTX 950 or higher
Storage	100 Mb of free space

Table 6.1: Hardware requirements

The game build supports Windows. We recommend 4 GBs of RAM to run the game smoothly, an i3 processor and a graphic card with enough power to play the game without frame drops.

6.2.2 Software

The game can be played without any external software. The only thing necessary is that the player will need an internet connection to be able to send the data.

Chapter 7

Studies and decisions

This chapter show the studies and decisions that we had been made during our investigation.

7.1 Unity3D and C#

I had experience with Unity3D. I had worked with the engine on various projects and some of them work with some of the basic game mechanics that we want to implement on this game, while some others had used data systems with Google's mail services to transfer data. With Unity3D we can forget about processing the whole game to see results in action. With Unity's 'Editor' presets we can create systems that works without the Editor running in 'Play' like the dungeon generator see in Figure 1.1

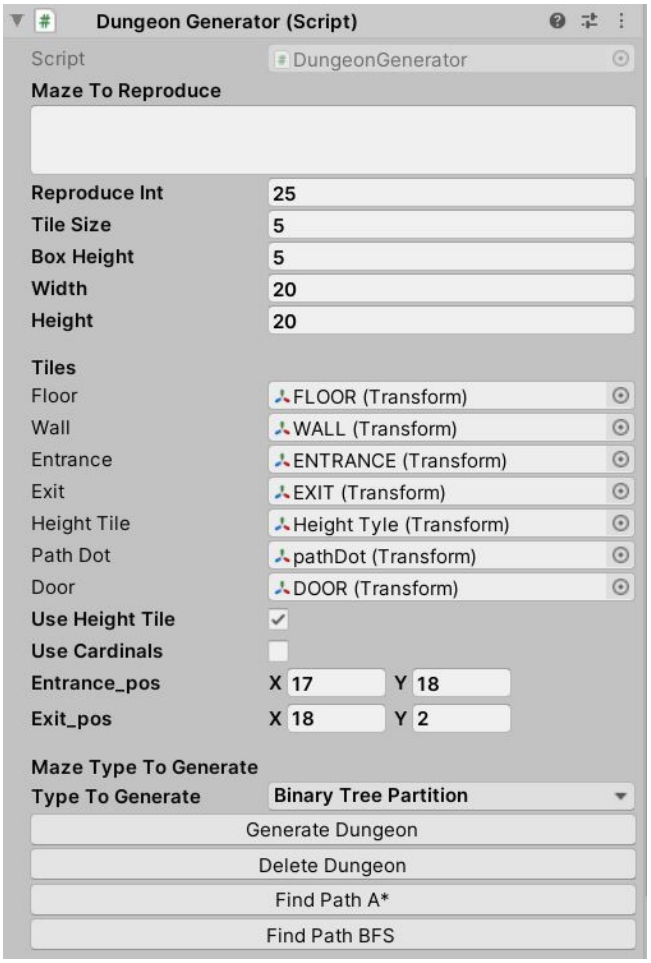


Figure 7.1: Dungeon Generator in Unity3D Editor

Unity3D provides 2 facilities for the development of this work, they are: the rendering and the builds. The rendering is used in both, the Unity3D Editor and the game rendering. The software allow us to edit the data chunks of the maze with 3D visual details and change everything from the position of the tiles to the colors and the textures, and fast test the results of those changes, unlike other frameworks like MonoGame [1].

7.2 Using Binary tree partition

The idea of creating a whole dungeon able to contains enemies, treasure chests, closed doors and keys, such as in Figure 7.2 [6], was an initial driver to create this project. The result of the method resembles the structure of Role-Playing Games genre like Zelda's games. This method is generally used in the rendering process of old First-person shooter games like Doom [2].

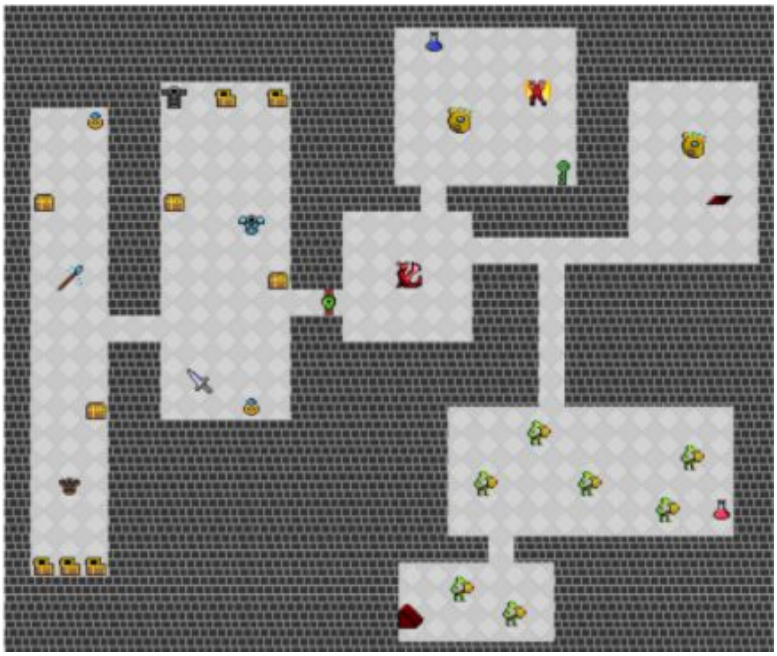


Figure 7.2: Example of a 2D dungeon created with BTP with rooms filled with enemies, chests and keys

We also use the BTP approach because it gave the most dungeon-like result of all the procedural maze algorithms we had studied. With this, we hope that in further work we could use this layout with other content creator algorithms to create a full level with the structure, enemies, rewards and many other game elements depending on the game genre, designer or the game scope.

7.3 Using A* and BFS

We use these algorithms because of their similarities between themselves and the data we could extract. Both algorithms share some structures in their code that we could use to in order to compare them with the player. We use BFS because of the similarity with the player behavior of not knowing where the exit is, and A* because it knows exactly how to get closer to the exit without opening a lot of nodes. This is the main base of our investigation.

[1] Free C# framework used to develop games for multiple platforms

Chapter 8

Analysis and design

For research purposes of this project, we establish two sections, the creation of a simple game in Unity3D to gather information from players about their performance and skills, and the analysis of the data gathered from playing the game and the correlation with the player steps in the maze and the pathfinding algorithms (A*, BFS). The game is divided into 4 main modules.

These are the main processes in which we are going to explain the functionalities of the game:

- Dungeon Generation Module
- Path Finding Module
- Data gathering and mailing Module
- Game Process Module

Each one of the modules works independent of another but together act as a whole system that generate levels, measure data and gather content of the player to help in this investigation.

8.1 Dungeon Generation

The dungeon generation process has 3 stages to create a maze: The initialization, generation and render.

```
2 references
public void GenerateDungeon()
{
    Initialize();
    Generate();
    Render();
}
```

Figure 8.1: Complete Dungeon Generation Process

In that specific order, the maze needs to be initialized making the whole area full of walls without any room or connection. Then, we proceed to generate the maze completely separated from the render module. We create the dungeon using the BTP process and we select the start and the exit points of the maze level. Finally we render the maze with the connections and the rooms created, along with the entrance and the exit.

8.1.1 Initialization

The initialization process, as its name indicates, initializes the data that we use in order to work with the map of the maze from scratch. Figure 8.2 shows how the maze is filled with wall tiles and the previous rendered maze is deleted. The render process is separated of the creation process so the deleted maze does not interfere with the current maze creation.

```
1 reference
void Initialize()
{
    Debug.Log("Initializing...");
    Debug.Log("Fixing Width & Height...");
    fixedWidth = (int)width;
    fixedHeight = (int)height;

    Debug.Log("Setting the dungeon...");
    maze = new int[fixedWidth, fixedHeight];
    for (int y = 0; y < fixedHeight; y++)
        for (int x = 0; x < fixedWidth; x++)
            maze[x, y] = (int)Tile.WALL;

    DeleteDungeon();

    mapHolder = new GameObject(holderName).transform;
    mapHolder.SetParent(transform);
}
```

Figure 8.2: Dungeon Generation Process Initialization

We set to an empty sate the A* data as we need to re-run the algorithm in every new maze to complete the whole creation process.

8.1.2 Generation

We create our maze using the Binary Tree Partition structure (BTP) to split the maze recursively and to create connected rooms. Figure 8.3 shows the process of generation. We can see 2 other maze generation algorithms that were tested before we decide to use BTP.

```
1 reference
void Generate()
{
    switch (typeToGenerate)
    {
        case MazeType.Prims:
            var prim = new Prims();
            prim.Generate(ref maze, fixedWidth, fixedHeight);
            break;

        case MazeType.DephtFirstSearch:
            var dfs = new DepthFirstSearch();
            dfs.Generate(ref maze, fixedWidth, fixedHeight);
            break;

        case MazeType.BinaryTreePartition:
            btp = new BinaryTreePartition();
            btp.Generate(ref maze, fixedWidth, fixedHeight);
            break;
    }

    Set_Entrance_Exit(ref this.maze);
}
```

Figure 8.3: Dungeon Generation Process Generation

With this code, we can generate different mazes, see Figure 8.4, but in this investigation we are going to stay with the BTP approach.

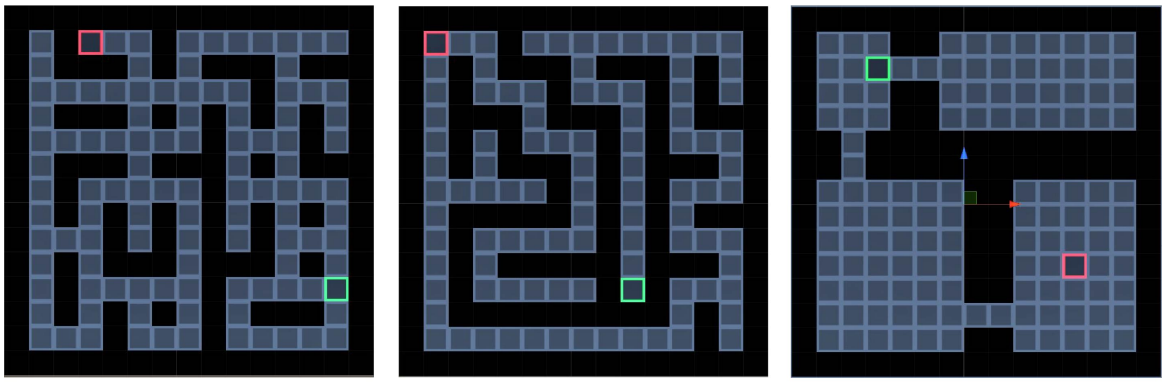


Figure 8.4: Comparation of different algorithm results

DFS creates easy mazes in were the exit is just walk through long corridors. Prim's algorithms creates path intersections and the structure of the maze is does not allow us to put new content on it. Both, DFS and Prim's are often used in 2D where the player can see thw whole maze and choose with a great knowledge of the level, but since our game will be 3D with a First-person camera, BTP proves to be a better option to the player experience.

We can create different rooms with different variations of BTP, with the same dungeon size. We show in Figure 8.5 an example of this.

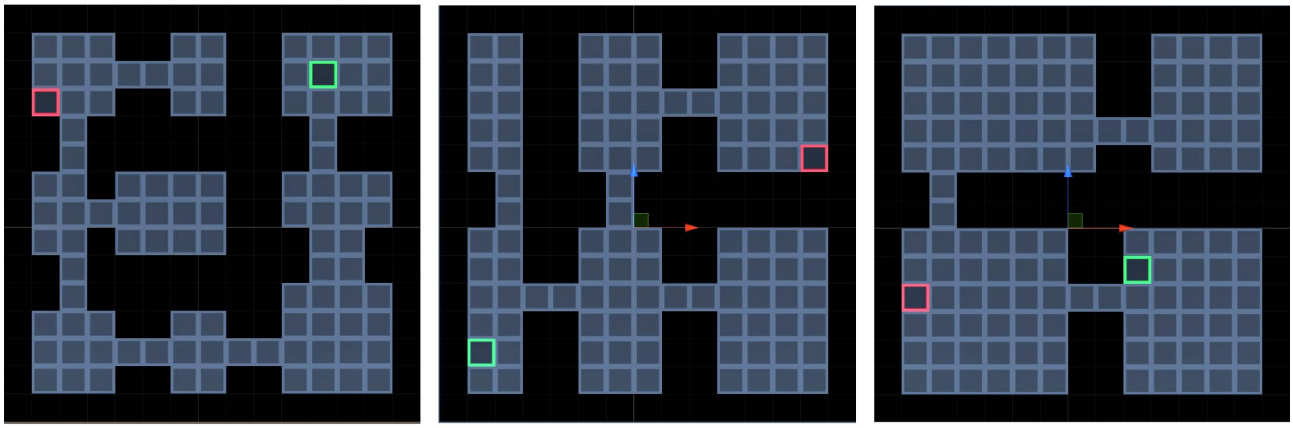


Figure 8.5: Comparative of different maze partition minimum size, the maze 'a' is 3, 'b' is 4 and the maze 'c' is 5

We finish the dungeon generation process by creating the entrance and the exit of the maze. Our approach to do it is using the size of the current maze; we use the half of it and set it as the minimum distance that the start and exit points must have in order close the maze generation process. Figure 8.6 shows the process of entrance and exit creation.

```
2 references
public void Set_Entrance_Exit(ref int[,] _maze){
    var pahtCount = 0;
    do{
        CreateEntrance();
        CreateExit();

        var path = new List<Vector2Int>();
        astar.FindPath(ref _maze, fixedWidth, fixedHeight, entrance_pos, exit_pos, out path, false, true);

        pahtCount = path.Count;

    }while(pahtCount < fixedWidth / 2);

    _maze[entrance_pos.y,entrance_pos.x] = (int)Tile.ENTRANCE;
    _maze[exit_pos.y,exit_pos.x] = (int)Tile.EXIT;
}
```

Figure 8.6: Creation of the entrance and the exit of the maze

The A* algorithm is used to know how far from the exit the start point is, and we compare to our minimum distance to make sure it meets the requirement. If not, we keep repeating the process until we complete it. We analyze it further in Section Pathfinding of this chapter.

8.1.3 Render

With the maze completely generated, we proceed to use Unity3D functions to create the walls and this floor of the maze. Finally, we traverse an array and interpret every tile number into a Unity's pre-edited 'GameObject'. Figure 8.7 shows the code. We use the function "Instantiate", which is useful to create objects in the game scene.

```
3 references
void Render()
{
    for (int y = 0; y < fixedHeight; y++)
    {
        for (int x = 0; x < fixedWidth; x++)
        {
            Vector3 position = CoordToPos(x, y);
            if (useHeightTile && (maze[y,x] == (int)Tile.WALL))
            {
                Transform newTile = Instantiate(heightTile, position, Quaternion.identity) as Transform;
                newTile.SetParent(mapHolder);
                newTile.localScale = tileSize * Vector3.one + Vector3.up * boxHeight;
            }
            else
            {
                Transform prefab = floor;
                switch (maze[y,x])
                {
                    case (int)Tile.FLOOR: prefab = floor; break;
                    case (int)Tile.ENTRANCE: prefab = entrance; break;
                    case (int)Tile.EXIT: prefab = exit; break;
                    case (int)Tile.WALL: prefab = wall; break;
                    case (int)Tile.DOOR: prefab = door; break;
                }

                Transform newTile = Instantiate(prefab, position, Quaternion.Euler(90, 0, 0)) as Transform;
                newTile.SetParent(mapHolder);
                newTile.localScale = tileSize * Vector3.one;
            }
        }
    }
}
```

Figure 8.7: Maze rendering process

In the process, we transform the 'x' and 'y' coordinates of the array into world coordinates that we use when we instantiate a GameObject tile. Then, the created object is sorted into a folder Object in the Unity hierarchy. With this, we render the floor, walls, entrance and exit tiles of the maze. Figure 8.8 shows a set of different maze results.

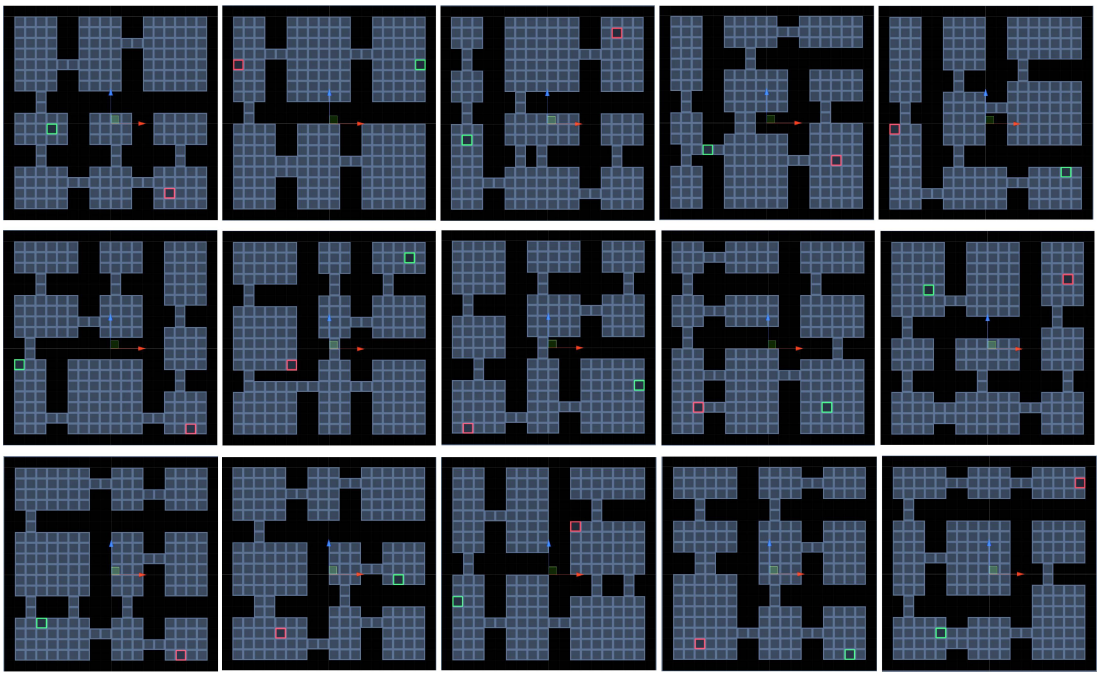


Figure 8.8: Diferent Maze Creation Process Results

The script can also include a height parameter to transform our current 2D plane map into a fully 3D finished map, as we can appreciate in the Figure 8.9.

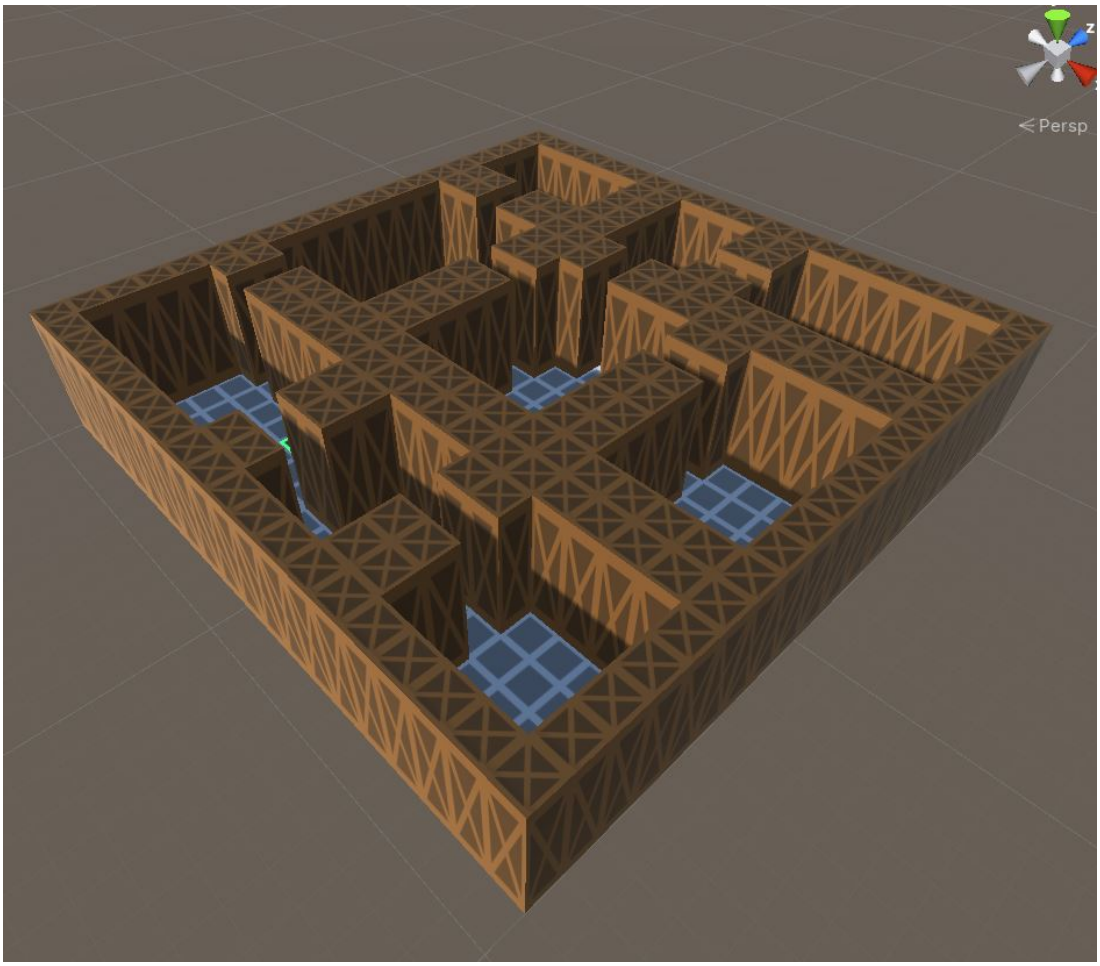


Figure 8.9: 3D Maze

In the game, we use procedural content generation to create multiple labyrinths. The player's objective is to find the exit in as fast as possible. In total there are 16 mazes: 4 mazes of size 10x10, 3 of size 15x15, 3 of size 20x20, 3 of size 25x25 and finally 3 of size 30x30. All the mazes created with a minimum room size of 3 and with complete collision with the floor and the walls. We try to gather the largest amount of player data without boring the player in the process. For this, we narrow the levels and make repetitions of size to appreciate if the player struggles in the same size or if the difficulty is in the change of the map.

8.2 Pathfinding

As we have commented in previous chapters, we use two pathfinding algorithms: A* and BFS. Both are very similar, sharing a resembling structure with the notable difference of the heuristic usage from A* algorithm. In our work, we first implemented A* because, to implement BFS, we just need to avoid using the heuristic in the A* code. We detailed this in Section Pathfinding: BFS from this same chapter. In Figure 8.10 we can see a maze generated of size 20x20.

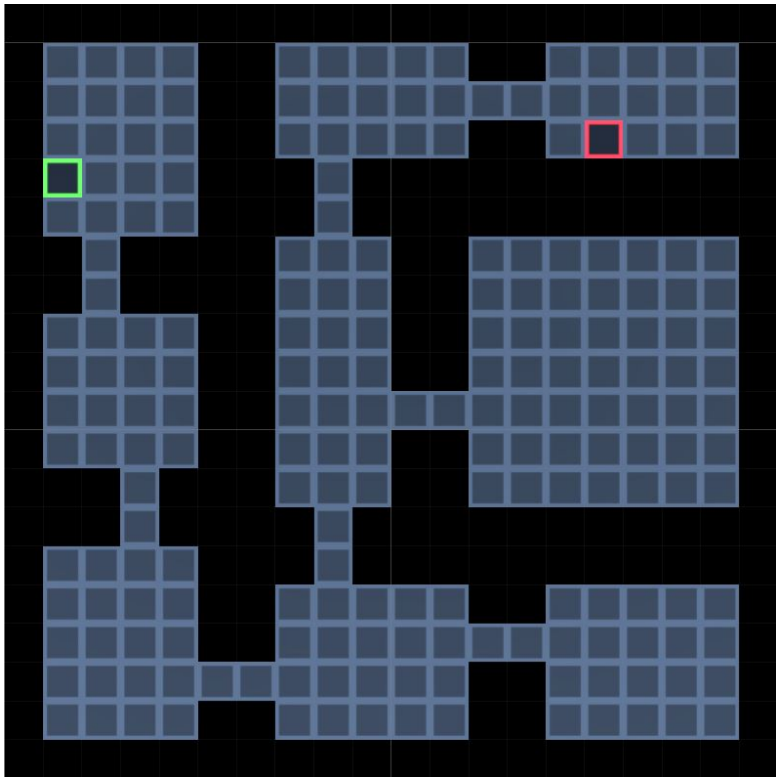


Figure 8.10: Example map of size 20x20

In this maze we are going to run A* and BFS algorithms, and we are going to compare the results of both implementations.

8.2.1 Pathfinding: A*

Once we finish our labyrinths, our next task was to implement A* into our maze and save the information of the optimal path and the open and closed lists, see Figure 8.11.

```
4 references
public class AStar
{
    // G cost = distance from starting node
    // H cost = distance from end node (heuristic)
    // F cost = G + H

    12 references
    public Node[,] maze;
    5 references
    public int width;
    5 references
    public int height;

    4 references
    List<Node> path;
    5 references
    HashSet<Node> closed;
    14 references
    List<Node> open;

    3 references
    public int OpenedNodes{get{return open.Count;}}
    2 references
    public int ClosedNodes{get{return closed.Count;}}
```

Figure 8.11: A* script variables

The node class contains the necessary logic to implement the A* algorithm, so the only structure we need modify so is the usage of a list to iterate through all the nodes in the map. As we can see in Figure 8.12, the variable 'f' of the class is a property which is calculated every time, so we only care about updating the variable 'h' (heuristic value) and 'g' (movement cost) every time we check the neighbor nodes.

```
28 references
public class Node
{
    11 references
    public int x;
    12 references
    public int y;

    5 references
    public int g;
    4 references
    public int h;

    4 references
    public int F { get { return g + h; } }

    2 references
    public bool isObstacle;

    2 references
    public Node parent;

    1 reference
    public Node(int _x, int _y, bool obstacle)
    {
        x = _x;
        y = _y;

        isObstacle = obstacle;
    }
}
```

Figure 8.12: A* Node class

The algorithm was split into two parts, the calling and the procedure. The calling was made to return the validation of the maze and the information about the optimal path that were used in-game to complete the map creation. Figure 8.13 shows how this public function works.

```
4 references
public bool FindPath(ref int[,] _maze, int w, int h, Vector2Int start, Vector2Int end,
out List<Vector2Int> _path,
bool cardinalOnly = true,
bool useHeuristic = true)
{
    path = new List<Node>();
    MazeToNodes(ref _maze, w, h);
    bool result = Astar(maze[start.y,start.x], maze[end.y,end.x], cardinalOnly, useHeuristic);
    _path = NodeListToVector(path);
    return result;
}
```

Figure 8.13: A* public method to be called in-game

By default, we use a cardinal representation (4 directions) of the map navigation and the heuristic calculations. We start by resetting any previous calculated path and creating a node

map representation of the tiles matrix. Then we call the A* implementation to proceed with the algorithm. We see in Figure 8.14 the header of the function and its parameters, the beginning of the function and the initialization of the open and closed lists.

```
1 reference
bool Astar(Node start, Node goal, bool cardinalOnly, bool useHeuristic)
{

    // OPEN //Nodes to be evaluated
    open = new List<Node>();

    // CLOSED //Nodes already evaluated
    closed = new HashSet<Node>();

    // add start node to OPEN
    open.Add(start);
```

Figure 8.14: A* method header and initialization

As we saw in the Algorithm 2, the implementation begins with the initialization for the lists and then we proceed with the cycle until there are no nodes left in the open list. We start at the root node, as we keep navigating the maze with the algorithm and gathering more nodes to the open list, while we search for the node with the lowest 'f' value, which represents the the sum of the h value and the g value. Figure 8.15 shows this process.

```
while (open.Count > 0)
{
    // current = node in OPEN with lowest Fcost
    var current = open[0];
    for (int i = 1; i < open.Count; i++)
    {
        if (open[i].F < current.F || open[i].F == current.F && open[i].h < current.h)
        {
            current = open[i];
        }
    }

    // remove current from OPEN
    open.Remove(current);
    // add current to closed
    closed.Add(current);

    if (current == goal)
    {
        RetracePath(start, goal);
        return true;
    }
```

Figure 8.15: A* code implementation, part 1

Once we get our current node to be expanded, we remove it from the open list and add it

to the closed one. Then, we proceed to check if the node is the exit of the maze or the goal node. If it is, we retrace the path from the exit to the start point. If it is not the goal node, we proceed to the next steps of the algorithm.

This is the final code that also includes the BFS algorithm, in section Pathfinding: BFS we explain which parts are critical in order to implement the algorithm using the same image seen in Figure 8.16.

```
    var neighbours = GetNeighbours(current, cardinalOnly);
    foreach (Node neighbour in neighbours)
    {
        //if (neighbour is not traversable)
        if (neighbour.isObstacle)
            continue;

        var addOn = (useHeuristic ) ? Manhattan(current, neighbour) : 0;
        int newMovCost = current.g + addOn;
        Node storedClose = Consider(closed, neighbour);
        if(storedClose != null){

            if(newMovCost < storedClose.g){
                closed.Remove(storedClose);
            }else{
                continue;
            }
        }else{
            Node storedOpen = Consider(open, neighbour);
            if(storedOpen != null){

                if(newMovCost < neighbour.g){
                    open.Remove(storedOpen);
                }else{
                    continue;
                }
            }
        }

        neighbour.g = newMovCost;
        if(useHeuristic)
            neighbour.h = Manhattan(neighbour, goal);
        neighbour.parent = current;

        open.Add(neighbour);
    }
}
return false;
```

Figure 8.16: A* code implementation, part 2

We start by obtaining the neighbors of the current node. If the cardinal only option is set to true, we avoid the diagonal neighbors. If not, we add them to the neighbors list. Then, for

every neighbor node of the current one, we check if it is an obstacle, in which case we ignore it. If not, we proceed with the calculation of the g value. We calculate the heuristic and we add it to the g value. We then go to check if the neighbor node is in the closed list. If the calculated g value is less than the value already computed in the same node, we remove it from the list and add it to the open list at the end of the process. This can be labeled as a node update. If it is not the case, we check if the node is in the open list. If is in it we then proceed to check if the calculated g value is less than the value stored in the open list. If it is true, we remove it from the list and update it with a new value, just like before.

Once we understand the overall function, we are going to explain a little bit further the heuristic usage and what is the equation that we use to calculate it. We specifically use the Manhattan distance, a very straightforward geometrical function in which we calculate the distance between two points in a 2D plane. We use this heuristic instead other like Octile distance becuase our algorithm does not move diagonally and the Octile heuristic is a better approach when the algorithms allow this movement. In Figure 8.17 we appreciate how easy is to implement this heuristic and its use in A* reduces the quantity of nodes opened.

```
2 references
int Manhattan(Node p, Node q)
{
    return Mathf.Abs(p.x - q.x) + Mathf.Abs(p.y - q.y);
}
```

Figure 8.17: Manhattan distance

Finally, with the implementation complete, we render the path as a demonstration of the algorithm functionality in Figure 8.18, with the map of Figure 8.10.

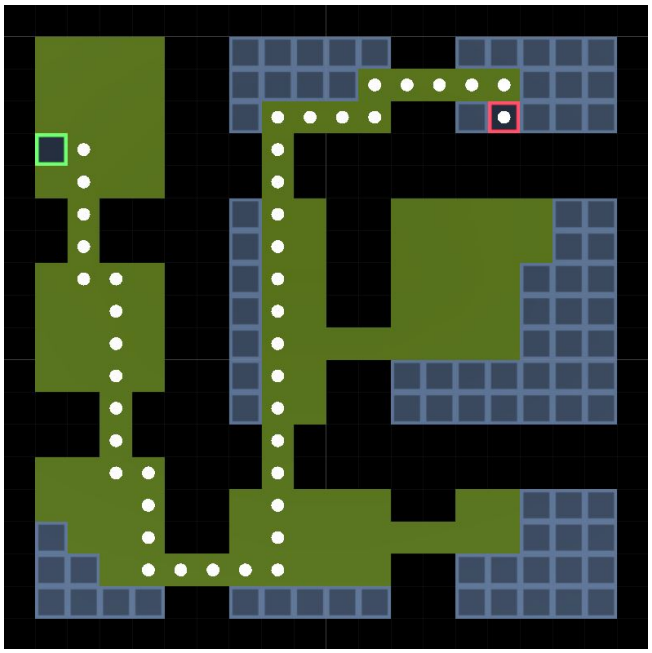


Figure 8.18: Path found with A* rendered in the map created. The green square is the entrance of the maze and the red one is the exit. The white dots represents the optimal path and the yellow squares represent the opened nodes of the algorithm

8.2.2 Pathfinding: BFS

With the A* implementation finished, the BFS implementation was a simple task. We are going now to show which parts of the algorithm are crucial to implement it with just a few adjustments to the A* code. The variables 8.11, node class 8.12 and the public function 8.13 are not changed or modified, those remain as they where defined in the A* implementation. We place our attention on the parameter 'useHeuristic' of the 'Astar' function. This is important due to the BFS not using the heuristic value to navigate through the nodes. If we do not use the heuristic, all the nodes that will be expanded will have the same value, so we are not going to choose the node with the smallest F value because they will be all at the same in the open list.

```
    var neighbours = GetNeighbours(current, cardinalOnly);
    foreach (Node neighbour in neighbours)
    {
        //if (neighbour is not traversable)
        if (neighbour.isObstacle)
            continue;

        var addOn = (useHeuristic ) ? Manhattan(current, neighbour) : 0;
        int newMovCost = current.g + addOn;
        Node storedClose = Consider(closed, neighbour);
        if(storedClose != null){

            if(newMovCost < storedClose.g){
                closed.Remove(storedClose);
            }else{
                continue;
            }
        }else{
            Node storedOpen = Consider(open, neighbour);
            if(storedOpen != null){

                if(newMovCost < neighbour.g){
                    open.Remove(storedOpen);
                }else{
                    continue;
                }
            }
        }

        neighbour.g = newMovCost;
        if(useHeuristic)
            neighbour.h = Manhattan(neighbour, goal);
        neighbour.parent = current;

        open.Add(neighbour);
    }
}
return false;
```

Figure 8.19: 'useHeuristic' parameter used in the function

The parameter is only used in the second part of the code implementation in Figure 8.16. We can see it detailed in Figure 8.19. Those are the sections affected by the heuristic. The heuristic value will be added to the g of the current node, but in this case we are using the BFS

algorithm, so we add it with value 0, so the g value of all the neighbor nodes will be exactly the same. The algorithm goes through all the nodes independent of the order, it will search around all of them after going through the next ones.

With those changes, the algorithm A* is modified and becomes a BFS. The results can be observed the same way as in the A* renderer path, in Figure 8.20.

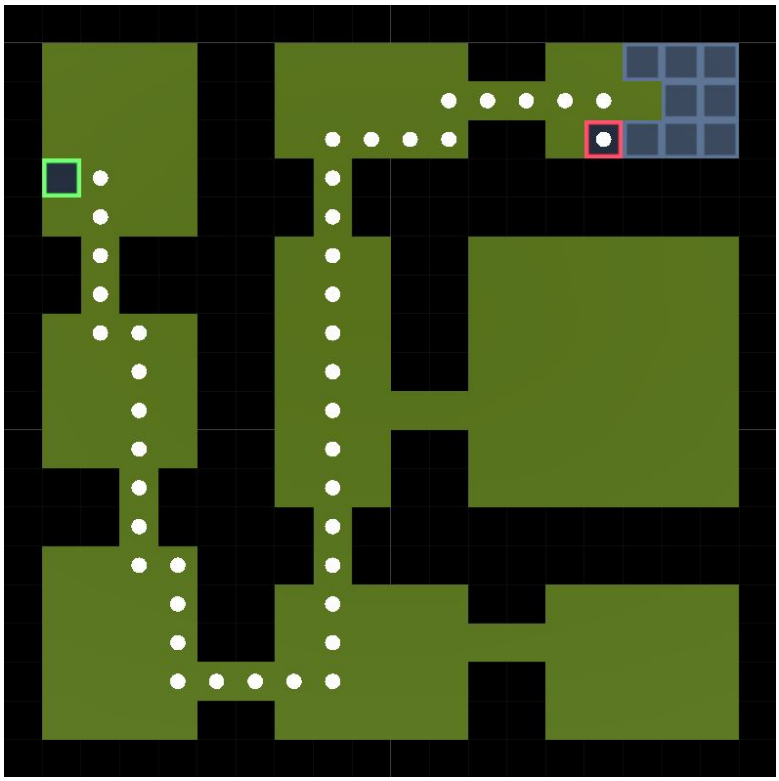


Figure 8.20: Path found with BFS rendered in the map created. The green square is the entrance of the maze and the red one is the exit. The white dots represents the optimal path and the yellow squares represent the opened nodes of the algorithm

The optimal path will be the same as seen Figure 8.18, but with the difference of the opened nodes, as shown in Figure 8.20. The opened nodes of the A* are less than the BFS because of the heuristic value, which gives the information of where the exit is.

8.3 Game Process

Once we have our dungeon creator system and pathfinding algorithms working, we proceed to create the game logic. The game characteristics that we need are the following:

- Player Movement
- Player Camera
- Game loop
- Menu & Settings

8.3.1 Player Movement

The player movement is based on First-Person Shooter games, were we move our character with the keyboard and the camera with the mouse. The movement in the X (Right-Left) and Z (Forward-Backward) axis implementation is simple and is done with a few lines of code. The Figure 8.21 shows how the movement in the tree axis works.

```
// Update is called once per frame
0 references
void Update()
{
    isGrounded = Physics.CheckSphere(groundChecker.position, groundDistance, groundLayerMask);
    if (isGrounded && velocity.y < 0)
    {
        velocity.y = -2f;
    }

    Vector3 move = transform.right * _directionalInput.x + transform.forward * _directionalInput.y;

    characterController.Move(move * Time.deltaTime * speed);

    velocity.y += gravity * Time.deltaTime;

    characterController.Move(velocity * Time.deltaTime);
}
```

Figure 8.21: Player movement script

We need to receive the inputs of the keyboard in order to move the player character in the game. We proceed to calculate the velocity with the speed parameter. We multiply the velocity with the direction of the movement and finally we check if the character is on the ground to perform the fall movement.

8.3.2 Player Camera

With the movement completed, we continue with the camera control. For this, we need to do as we have done earlier with the movement of the player to receive the inputs, but this time inputs from of the mouse, as seen in Figure 8.22.

```
0 references
void Update()
{
    sensitivity = PlayerPrefs.GetFloat(SettingsMenu.mouseTag);
    float mouseX = _directionalLook.x * sensitivity * Time.deltaTime;
    float mouseY = _directionalLook.y * sensitivity * Time.deltaTime;

    xRot -= mouseY;
    xRot = Mathf.Clamp(xRot, -90, 90);

    transform.localRotation = Quaternion.Euler(xRot, 0, 0);
    player.Rotate(Vector3.up * mouseX);
}
```

Figure 8.22: Camera movement script

Both the camera and the movement receive the inputs from another class that is in charge of the input management of the player, and does not interfere with the camera movement or the player movement behavior, only with its directional inputs, as we see in Figure 8.23.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

0 references
public class PlayerController : MonoBehaviour
{
    2 references
    public PlayerInputHandler playerInput;
    1 reference
    public PlayerMovement playerMovement;

    1 reference
    public PlayerCamera playerCamera;

    0 references
    void Update()
    {
        playerMovement.SetDirectionalInput(playerInput.DirectionalInput);
        playerCamera.SetDirectionalLook(playerInput.DirectionalLook);
    }
}
```

Figure 8.23: Player input script

Finally, our character behavior is ready. We add a visual effect that shows dots where the player was for a short time, so he/she knows in which room the player was before. Figure 8.24 shows the player hierarchy and Figure 8.25 shows the player structure in the game. All our player logic is in the "Player FPS" object that controls both the camera and the player movement. The "GroundChecker" object checks if the player is touching the ground or not. The "Cylinder" is the collider of the player and the "Particle System" is the visual effect that shows the last steps of the player.

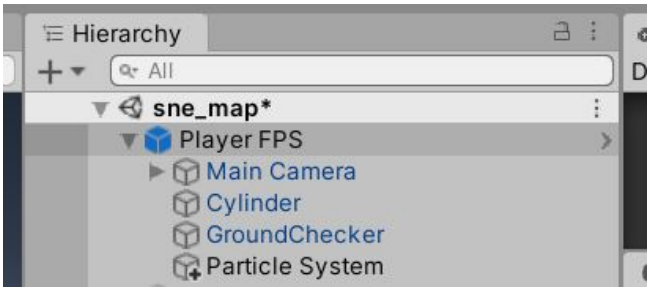


Figure 8.24: Player hierarchy

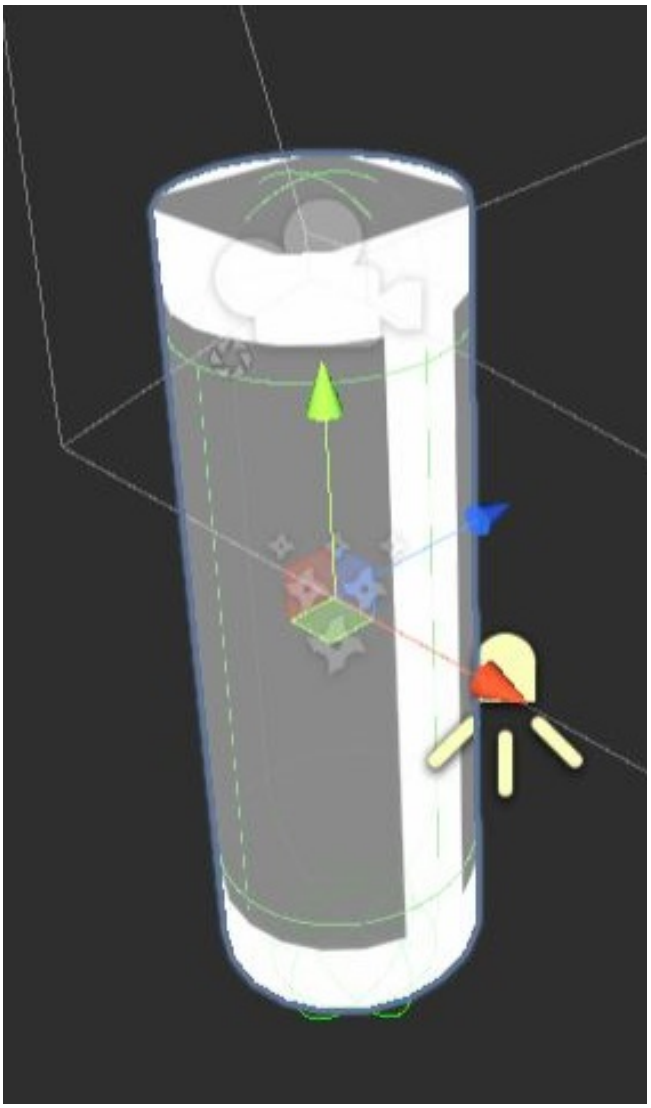


Figure 8.25: Player structure

As we can see in the player structure in the game, we do not use a full model or animations

since our main goal is to realize an investigation of the player behavior and not to create a commercial game, at least yet.

8.3.3 Game loop

The "GameManager" class is the controller of all the things that occur in the game and is in charge of beginning and finishing the game processes, such as to start the game, reset the levels and finish the game process. With this in mind, the game manager is the one that controls the dungeon creator system and stays alert about the player behavior in all of the game process.

The game uses a time trial, where we have to pass trough a set of levels in the smallest time possible, so whenever we are left out of time, we lose the game. If we pass through all the levels of the game, we complete it. In both cases, we are going to request the player to send the data that we collected: the steps of each level, the maze layout and size.

8.3.4 Menu & Settings

There are three UI objects in the game, the main menu, the settings menu and the Head-up Display. The first one refers to the first menu and most interactable of them all. It gives three options: "Play" (to start immediately the game), "Settings" (to open a settings menu) and the "Exit option" (to close the game).

Figure 8.26: Main menu of the game

The Settings menu at, Figure 8.27, is the one in charge of giving the user the options of changing the resolution, the graphics, the volume of the music, the mouse sensibility and the options of playing on window or full screen resolution.

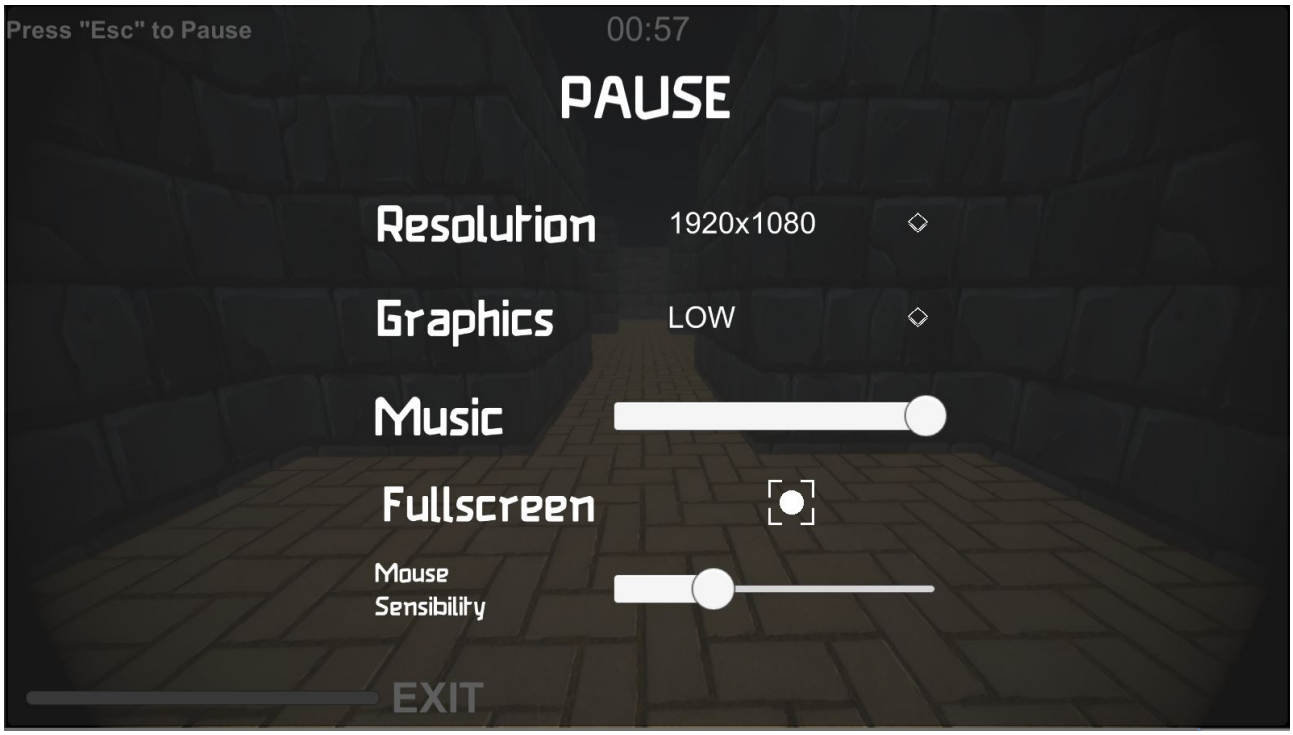


Figure 8.27: Settings of the game

The settings menu is also displayable in the main game scene with exact the same options layout. The HUD of the game shows how far the players are from finishing all the mazes and the time left to complete the game.

Figure 8.28: HUD of the game

8.4 Data gathering and mailing

The final part of our game is to be able to save the information into a convenient format and send it to us. We are going to gather the number of mazes completed by the player, along with the following information of every maze completed, as is shown in Figure 8.29.

```
[System.Serializable]
7 references
public class DungeonData
{
    //Player
    4 references
    [SerializeField] public int playersSteps;
    6 references
    [SerializeField] public float completitionTime;
    6 references
    [SerializeField] public int size;
    3 references
    [SerializeField] public string maze;
```

Figure 8.29: Maze data

The player steps are the nodes visited by the player through the level. Every time the player steps into a new tile, it is added to the counter. The completion time is the total time that the player took to complete the maze. The maze structure is the complete maze broken down into a single string data. Finally the size of the maze is also sent.

Once we save the raw data of the player's walkthrough, we proceed to show an end game screen in which the player would give his consent to use this data, as seen in Figure 8.30.

Figure 8.30: Send Data screen

Then the data is sent in a JSON format. We use our own mail to retrieve all the data, as shown in Figure 8.31.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

using System.Net.Mail;

1 reference
public static class MessageHelper
{
    1 reference
    public static void SendData(){
        string mailOrigen = "afrobotdev@gmail.com";
        string mailDestino = "afrobotdev@gmail.com";

        string password = "gynwmqpfrhvdeysh";

        string data = JSonParser.pathName;

        MailMessage message = new MailMessage(mailOrigen, mailDestino, "Recopilacion de Datos", "<b>Muchas Gracias por jugar</b>");
        message.IsBodyHtml = true;

        message.Attachments.Add(new Attachment(data));

        SmtpClient client = new SmtpClient("smtp.gmail.com");
        client.EnableSsl = true;
        client.UseDefaultCredentials = false;
        client.Port = 587;
        client.Credentials = new System.Net.NetworkCredential(mailOrigen, password);

        client.Send(message);
        client.Dispose();
    }
}
```

Figure 8.31: MessageHelper class that help us to retrieve the player data

Once we get the data, we use the pathfinding algorithms to retrieve more information of the mazes. We reproduce the map with the saved data and calculate the optimal path along with the open and closed lists of both the A* and BFS algorithm. To this end, we use a class

where we calculate these values and save them into a 'csv' format. Those are the initial data, together with the map structure and the pathfinding algorithm data implemented as variables in Figure 8.32.

```
7 references
public class CalculatedMaze{

    6 references
    public int playersSteps;
    4 references
    public float completitionTime;
    3 references
    public int size;
    5 references
    public int bfs_openedNodes;
    5 references
    public int bfs_closedNodes;
    3 references
    public int bfs_steps;
    4 references
    public int astar_openedNodes;
    4 references
    public int astar_closedNodes;
    5 references
    public int astar_steps;
    4 references
    public int whiteSpaces;
```

Figure 8.32: Complete player data

Figure 8.33 shows the maze reproduction and Figure 8.34 shows how the pathfinder algorithms data is saved.

```
2 references
public CalculatedMaze(int _playerSteps, float _completitionTime, string _mazeString, int _size){
    this.playersSteps = _playerSteps;
    this.completitionTime = _completitionTime;


    //Debug.Log(_mazeString);
    int[,] maze =  new int[_size,_size];
    Vector2Int start = Vector2Int.zero, exit = Vector2Int.zero;
    for(int x = 0; x < _size; x++){
        for(int y = 0; y < _size; y++){
            var index = y + (_size * x);
            maze[y,x] = int.Parse("" +  _mazeString[index]);

            if(maze[y,x] == (int)Tile.ENTRANCE){
                start = new Vector2Int(x,y);
            }

            if(maze[y,x] == (int)Tile.EXIT){
                exit = new Vector2Int(x,y);
            }

            //Espacios en Blanco
            if(maze[y,x] == (int)Tile.FLOOR){
                whiteSpaces++;
            }
        }
    }
```

Figure 8.33: Complete data calculation, reproduction of the maze

```
    //Llamar astar y bfs aqui
    AStar aStar = new AStar();

    //Astar
    var list = new List<Vector2Int>();
    aStar.FindPath(ref maze, _size, _size, start, exit, out list, false, true);
    this.astar_steps = list.Count;
    this.astar_openedNodes = aStar.OpenedNodes;
    this.astar_closedNodes = aStar.ClosedNodes;

    //BFS
    list = new List<Vector2Int>();
    aStar.FindPath(ref maze, _size, _size, start, exit, out list, false, false);
    this.bfs_steps = list.Count;
    this.bfs_openedNodes = aStar.OpenedNodes;
    this.bfs_closedNodes = aStar.ClosedNodes;

    switch(_size){
        case 11:
        case 21:
        case 31: _size -= 1; break;
    }
    this.size = _size;

    //System.Threading.Thread.Sleep(1000);
}
```

Figure 8.34: Complete data calculation, using the pathfinding algorithms to save the optimal path, closed and open lists

Then, finally, our whole set of data is ready to be analyzed. The data retriever is automatic, so we only need to download it from the email, were there are stored all the player's data, and transform the whole set into our "csv" format file.

Chapter 9

Deploying and testing

Once we got our system working on the Windows platform, we proceed to gather test subjects to play the game. To do so, we first tested the system our self, gathering our own data. We see in Figure 9.1 how we got the JSON format file into our mail account, which in this case works as a database.

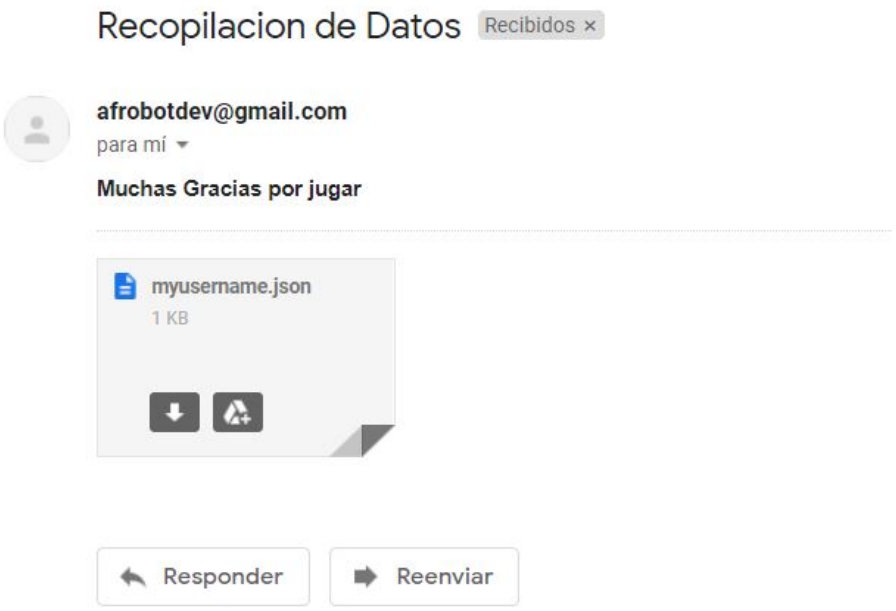


Figure 9.1: Test of the gathering data system

Figure 9.2 shows the content of the file. As we see we only need the size of the maze and the complete string of its structure to reproduce the labyrinth and calculate the complete data.


```
{
    "_levelsCompleted": 1,
    "_datas": [
        {
            "playersSteps": 12,
            "completitionTime": 7.910367965698242,
            "size": 11,
            "maze": "11111111111100011000011030000000110001100001110111000011101111011110001110111100011000011000110200110001100001111111111111"
        }
    ]
}
```


Figure 9.2: Example of the player's data in JSON format

The figure above represents just one maze completed of a set of 16 with different sizes. Once

we upload our game onto a web page called itch.io [1] and we retrieve various data from people worldwide. We ere able to gather information of people around the world with the help of Game Jams. Game Jams are game developing competitions were you are challenged to create a game with or without some limitations. Those jams help us into reaching out a great size of people to play our game.

We end up having 27 test subjects with 16 mazes completed each, giving us a total of 432 completed mazes data.

9.1 Problems

One of our main problems during the testing phase of the project was the lack of test subjects. Since the test phase started we been searching for different ways to gather data of players, but with no good results. We still want to reach 100 users, but we currently have only 30 test subjects.

COVID-19 pandemic and restrictions hit us hard on our schedule. Our origin plan was to present the project in July, but due to the global contingency we had to re-schedule and finish the work to present it in September.

Using `Python3` and the libraries `numpy` and `matplotlib` to create the graphs shown was a huge time consumer. We spent much time into learning the language and using `numpy` and `matplotlib` to the creation of the graphs. In the end, we use Excel as a fast tool to manipulate the data retrieved from the test subjects. The lack of experience into conducting investigation of this type was a huge setback, but a great learning opportunity.

[1] https://itch.io/

Chapter 10

Results

We created multiple graphs in search for relations between the player steps and completion time of every maze with the pathfinding data, such the optimal path and the closed and open lists of A* and BFS algorithms.

To this end, we use all of our current database. The final number of test subject is 30 and correspond to 426 complete mazes. With this data we start to calculate the average results of different data combinations. The Figure 10.1 shows one of those combinations that is the percentage of optimal paths that players use in the maze completed versus the nodes that the algorithm added to the open list by the A* algorithm.

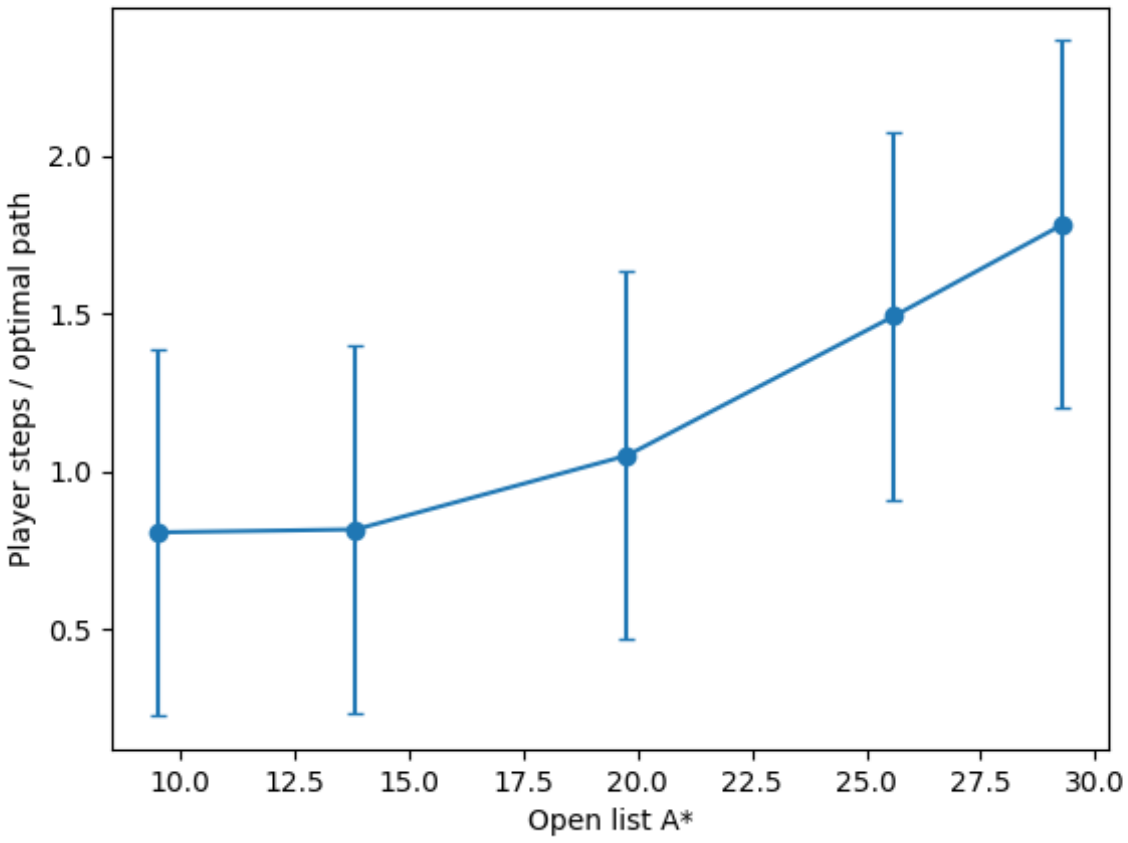


Figure 10.1: Percentage of the optimal path by the player vs the Open list of A*

The Y axis represents the **fraction over the optimal path used by the player** 10.1. The X axis is the size of the open list (nodes left not calculated) of the A* algorithm. The 5 points represent the mazes grouped by the maze sizes (10x10, 15x15, 20x20, 25x25 and 30x30). The vertical lines represent the error margin of those averages values.

The formula used to calculate the fraction over the optimal path used by the player is:

Equation 10.1 Fraction over the optimal path used by the player

$$x = \frac{player\_steps - optimal\_path}{optimal\_path} \tag{10.1}$$

We see that our x represent the fraction, which will always be a value that is larger than the optimal path.

We can observe how, in the first levels, the player always uses more than the 50% of the optimal path founded by A*. Let is say, if the algorithm founds a path that is in a distance of 7 nodes far from the entrance, the player will usually end up walking through 14 or more nodes. In the higher levels, this increase can be more than a 200% of the optimal path. This implies that the player gets more and more lost in every maze and it is progressively more difficult to find the exit. Another graph is shown in Figure 10.2, which represents the percentage of optimal path used, but this time with the closed list (opened nodes) of A*.

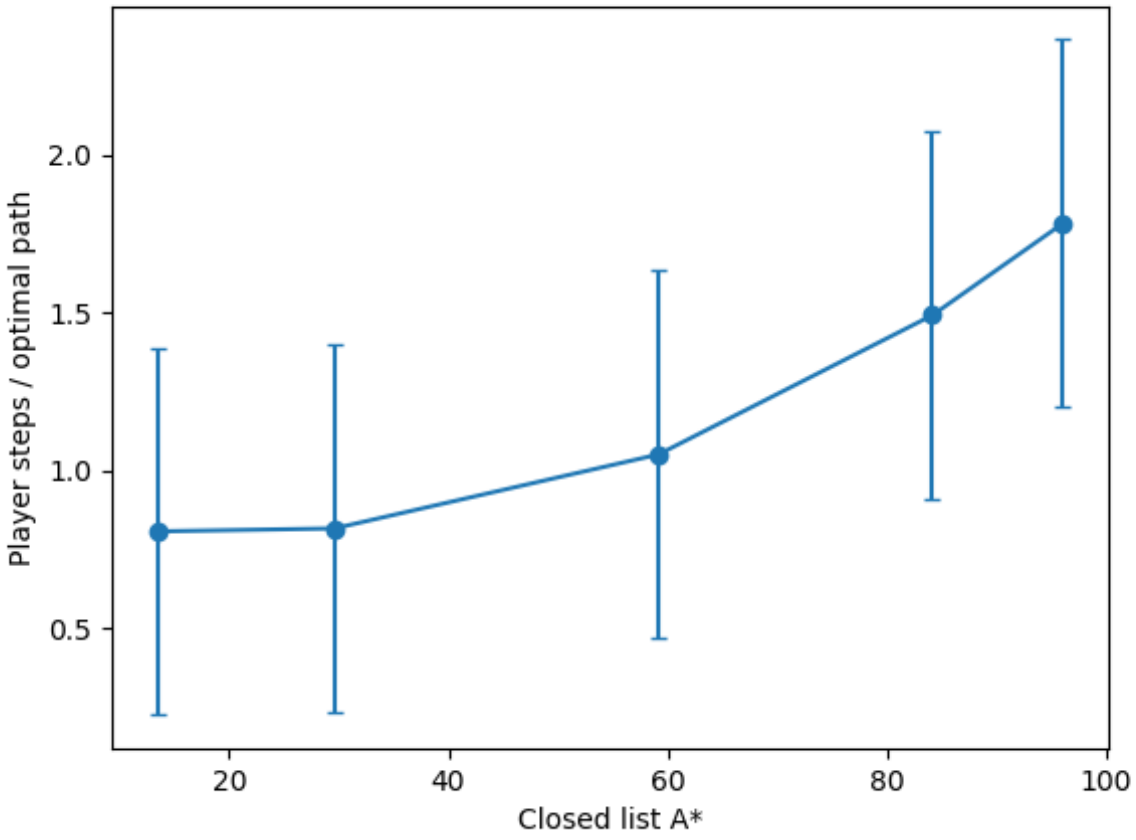


Figure 10.2: Percentage of the optimal used path by the player vs the Closed list of A*

The graph shows that the average size of the closed list remains inferior of 100% this is because of the heuristic used in the algorithm, which acts as an imaginary compass for the algorithm that is trying to reach the exit in the less steps possible. Now, we show in Figure 10.3 a graph in which we compare the percentage of use of the optimal path in the Y axis and the Closed list of the BFS algorithm in the X axis.

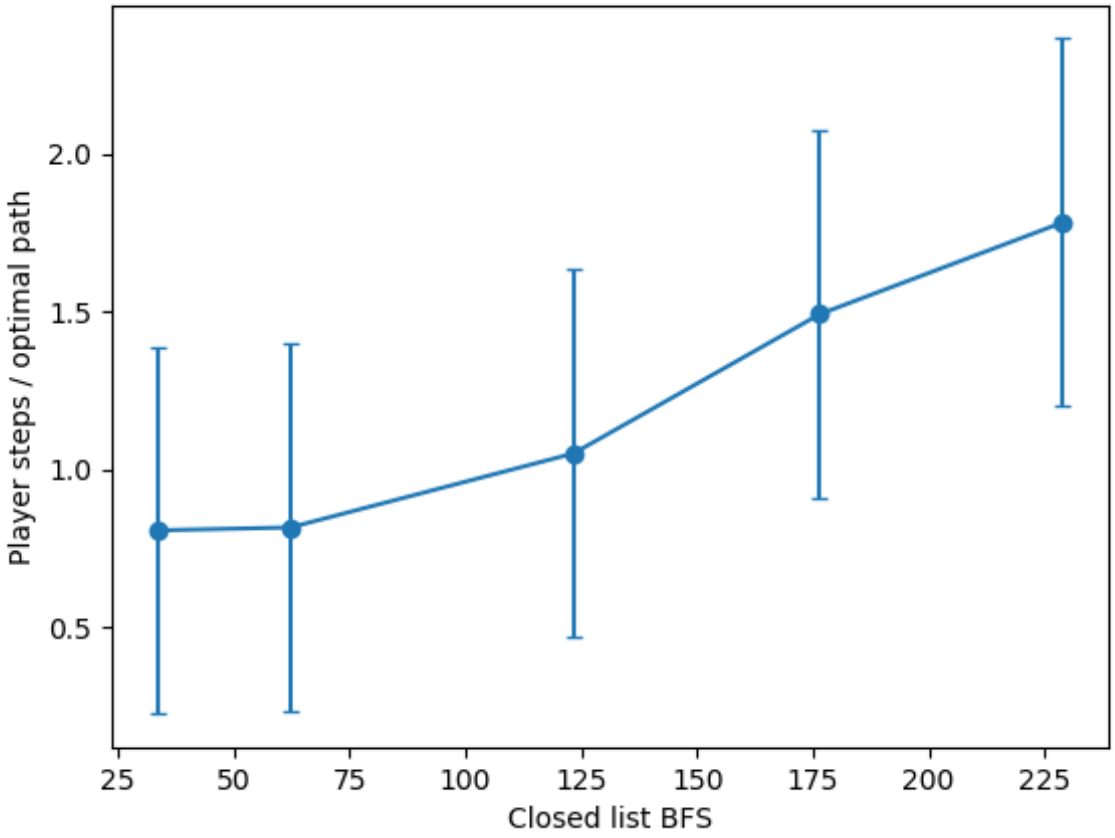


Figure 10.3: Percentage of the optimal path by the player vs the Closed list of BFS

The average of the evaluated nodes is higher than the previous graph. In this case, the algorithm is trying to reach the exit searching through all the possible nodes, the algorithm does not know where the exit is, and is searching around his area passing through all the nodes around the root node. The algorithm, as the player, is trying to find the exit without further knowledge of the right direction.

We analyzed how much the player walks around the map, and compared it to the "walkable" nodes of the map. Figure 10.4 shows the map percentage traversed by the player.

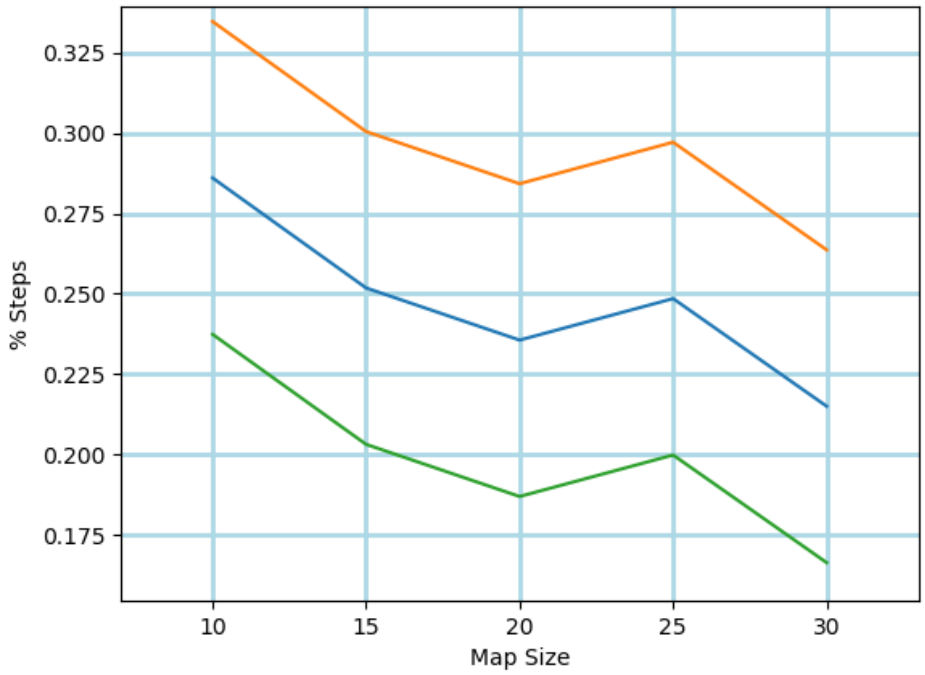


Figure 10.4: Walked map percentage, the blue line represents the mean values of the visited part of the map by the player, the green and orange lines represent the minimum and maximum error margin of each value. The graph shows that the players do not even walk the 50% of the complete map, they reach the exit before walking the entire maze, or even a great part of it

We take a further analysis into the opened nodes of the A* and BFS and compared to the steps the player takes. In Figure 10.5 we see the arithmetic and geometric means of the opened nodes (closed list) by A*.

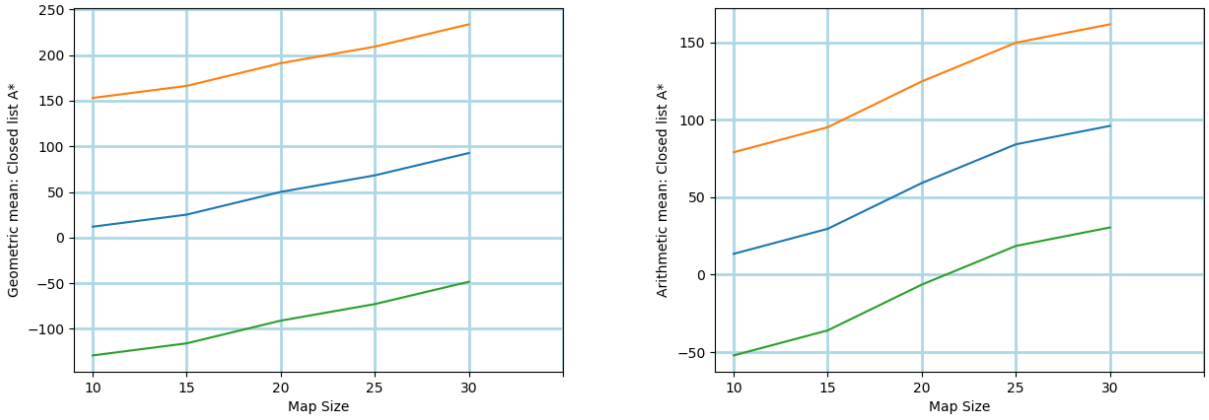


Figure 10.5

As in the previous graphs, we can see the blue line that represents the mean values and the green and orange lines that represent the minimum and maximum respectively. Both graphs are very similar, which means that in our data set there are no strange values far from the average. The error range seems wide in comparison to the arithmetic mean. This represents that our arithmetic mean has a smaller percentage of error in the measurement of the average

values of the nodes opened. Now, in Figure 10.6 we do the same but with the BFS opened nodes.

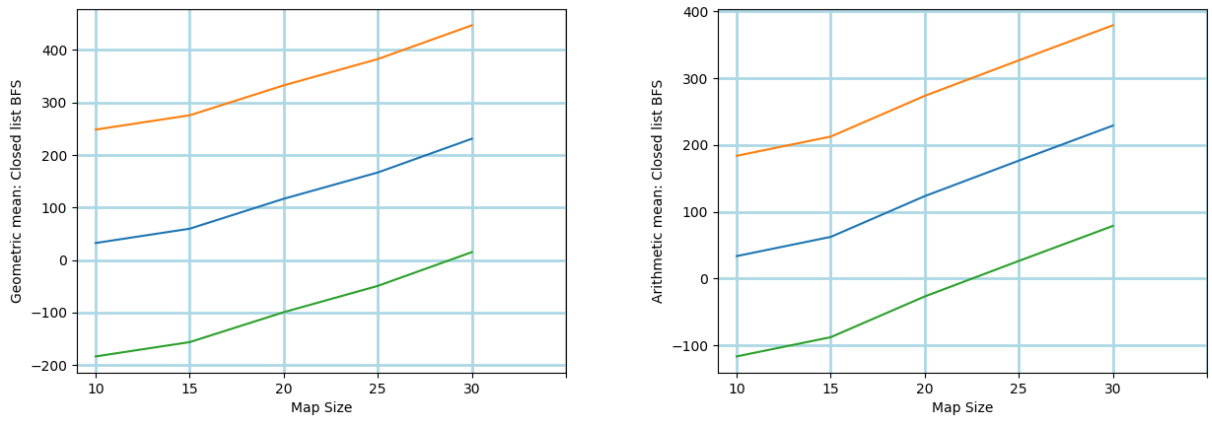


Figure 10.6

The blue line shows the mean values and the green and orange lines the minimum and maximum error values. We see the same behavior as in our previous graph 10.5, the mean value is almost the same and the error percentage of the geometric graph is wider that the arithmetic graph.

This is the main information about the player behavior. We analyzed further the relation between the player steps and the algorithm open and closed list with correlation coefficient, as we see a relationship between the BFS and A* opened nodes and the player steps. The results of Table 10.2 are the results of comparing the different values and correlations to determine the highest ones.

Variables	Correlation coefficient
playerSteps & ClosedBFS	0.67
playerSteps & Closed+Open BFS	0.68
completionTime & ClosedBFS	0.67
playerSteps & optimalPath	0.57
playerSteps & OpenBFS	0.3
worstTime & playerSteps	0.47
bestTime & playerSteps	0.33
mapSize & optimalPath	-0.74

Table 10.1: Correlation coefficient values

These are the best correlated values with the player steps. We see that the measure that is the most connected to the player steps is the closed and opened lists of BFS, with a value of 0.68, followed by the player steps with just the closed list of nodes.

We show the various approximations that we tested to find the values related with the player steps. The values were extracted from a larger list of possible values in search of the one that best represents the player's performance.

Correlaciones	size	player completion	Optimal	Open(A*	Closed(A*	Open(BF	Closed(BF	walkabl	level
size	1.00	0.54	0.58	0.73	0.74	0.61	0.37	0.82	0.98	0.02
player steps	0.54	1.00	0.81	0.57	0.51	0.57	0.30	0.67	0.57	-0.05
completion time	0.58	0.81	1.00	0.57	0.50	0.58	0.35	0.67	0.60	-0.09
optimal path	0.73	0.57	0.57	1.00	0.76	0.93	0.07	0.90	0.71	0.00
Open(A*)	0.74	0.51	0.50	0.76	1.00	0.68	0.17	0.84	0.73	-0.01
Closed(A*)	0.61	0.57	0.58	0.93	0.68	1.00	0.02	0.88	0.61	-0.01
Open(BFS)	0.37	0.30	0.35	0.07	0.17	0.02	1.00	0.23	0.37	0.00
Closed(BFS)	0.82	0.67	0.67	0.90	0.84	0.88	0.23	1.00	0.82	0.01
Open+Closed(A*)	0.66	0.59	0.59	0.94	0.77	0.99	0.05	0.91	0.66	-0.01
Open+Closed(BFS)	0.83	0.68	0.67	0.90	0.84	0.87	0.26	1.00	0.83	0.01
player steps - optimal path	0.45	0.99	0.78	0.43	0.41	0.44	0.32	0.56	0.48	-0.05
open(A*)/optimal path	-0.51	-0.33	-0.35	-0.74	-0.27	-0.65	0.02	-0.56	-0.46	-0.05
closed(A*)/optimal path	0.55	0.49	0.53	0.78	0.63	0.91	0.02	0.79	0.53	0.03
open+closed/	0.40	0.43	0.46	0.57	0.62	0.76	0.03	0.67	0.40	0.01
closed(BFS)/	0.66	0.48	0.53	0.34	0.58	0.34	0.47	0.67	0.66	0.03
optimal path/size	0.30	0.36	0.35	0.85	0.52	0.80	-0.19	0.63	0.28	-0.02
closed/ size	0.65	0.61	0.61	0.89	0.79	0.89	0.12	0.95	0.63	0.00
completion time/ player steps	-0.34	-0.31	-0.17	-0.32	-0.32	-0.29	-0.08	-0.34	-0.33	-0.19
(player steps - optimal path)/optimal	0.26	0.72	0.73	0.06	0.15	0.09	0.38	0.22	0.29	-0.05
player steps/walkable path	-0.09	0.66	0.53	0.14	0.05	0.21	-0.01	0.17	-0.07	-0.04
open+closed(A*)/walkable path	-0.26	0.06	0.05	0.35	0.14	0.47	-0.42	0.17	-0.27	-0.03
open+closed(BFS)/walkable path	-0.26	0.13	0.11	0.28	0.17	0.38	-0.21	0.24	-0.27	-0.01
closed(BFS)/walkable path	-0.13	0.20	0.18	0.40	0.28	0.48	-0.26	0.36	-0.14	-0.01
completion time/player steps x optimal	0.62	0.33	0.50	0.90	0.65	0.82	0.02	0.76	0.59	-0.04
completion time/player steps x	0.78	0.47	0.64	0.84	0.78	0.82	0.24	0.92	0.77	-0.02
open+closed(BFS)/size	0.63	0.62	0.61	0.88	0.78	0.88	0.16	0.94	0.62	0.00
open+closed(A*)/size	0.40	0.45	0.46	0.85	0.64	0.94	-0.13	0.75	0.38	-0.02

Figure 10.7: Correlation coefficient table

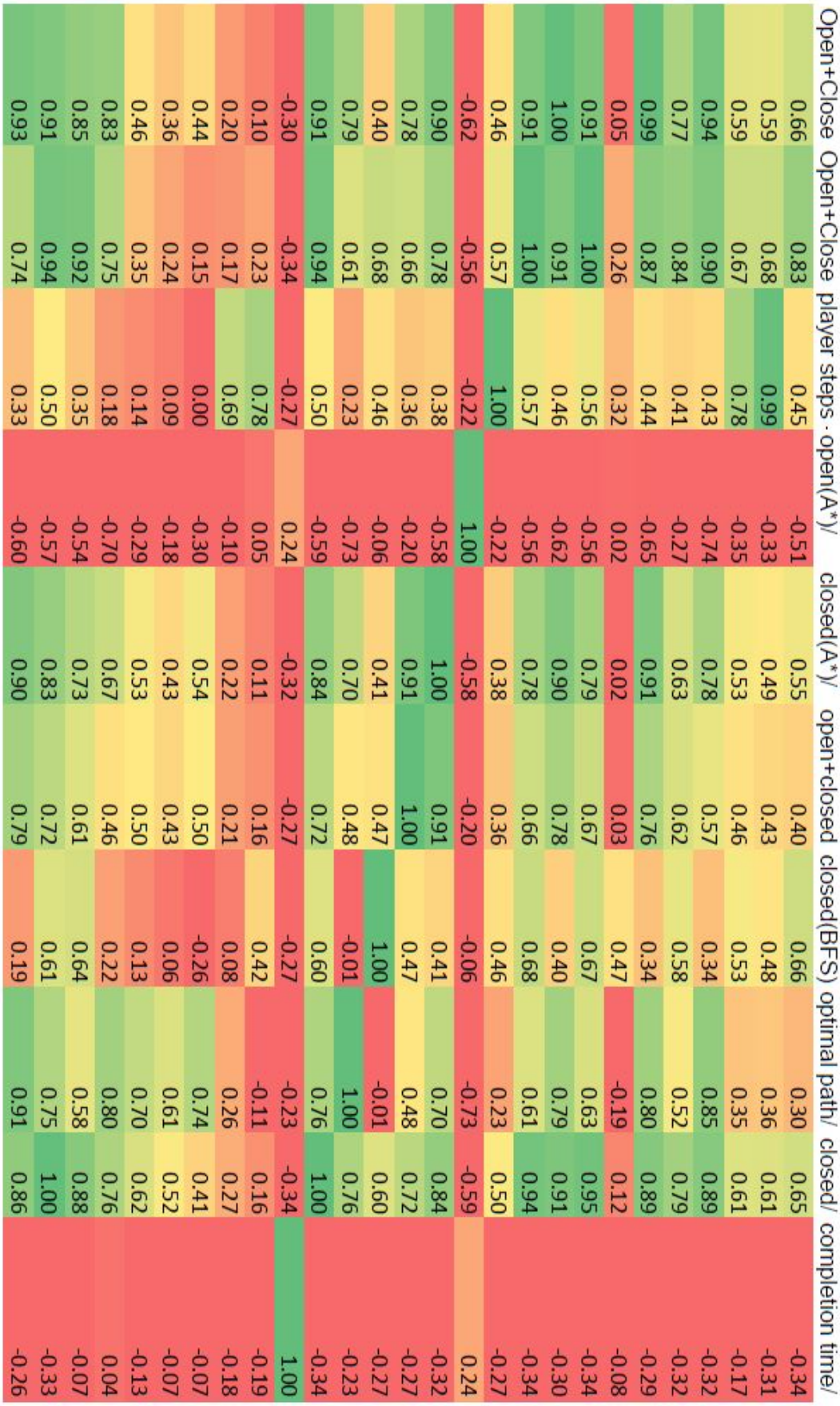


Figure 10.8: Correlation coefficient table

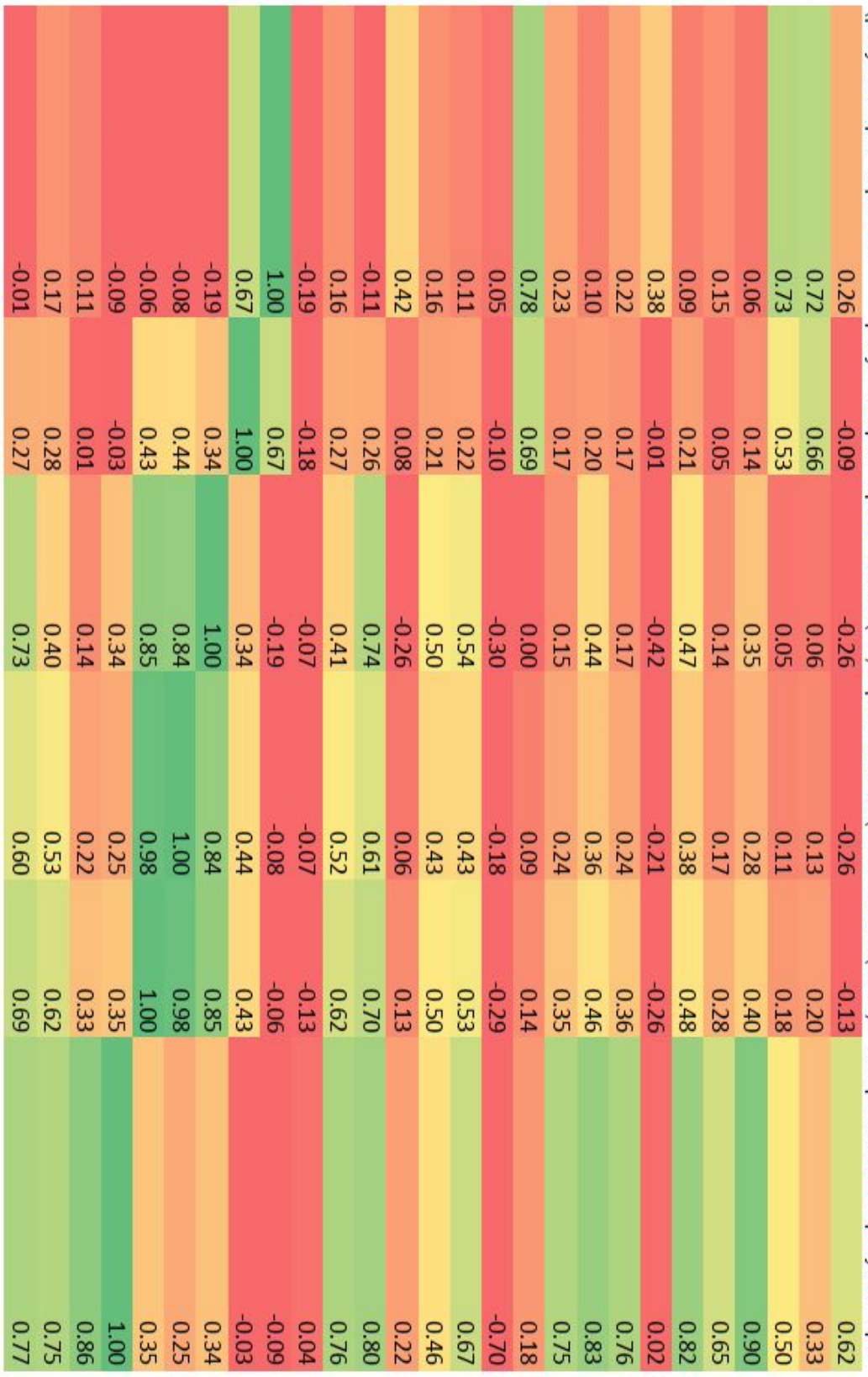


Figure 10.9: Correlation coefficient table

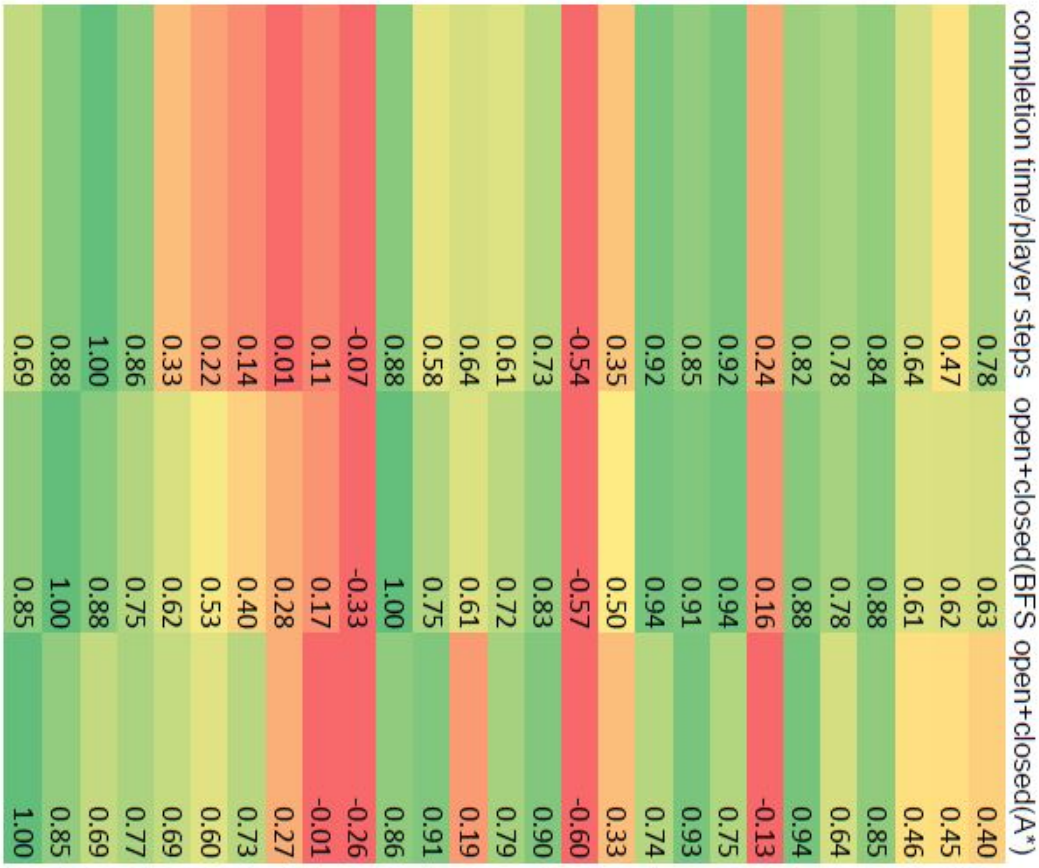


Figure 10.10: Correlation coefficient table

The table shows the our attempts to find the values that are related to the player steps, we try with equations and calculus and compare each one of them with one another. The results of those comparatives are in the table where a green value represent a strong relation between the values compared and a red value shows a weak or a negative relation with the compared value. Some of them are combinations of node lists (closed+open) of both A* and BFS, some other are fractions or percentages of the complete walkable space of the maze.

We conclude that the player steps are strongly related to the opened nodes (closed list) of the BFS algorithm because they have the same amount of information as the player about the map and search for the exit, so they have to "walk" through all the possible nodes until they find the exit.

We achieved all the initial tasks of the project and we plan to keep analyzing the player behavior as further work, as we are going to see in Chapter 12 of this report.

Task	Status
Investigation: DDA Systems	OK
Investigation: PGC Systems	OK
Task divition	OK
Binary Tree Partition implementation	OK
Procedural Dungeon implementation	OK
A* algorithm implementation	OK
BFS algorithm implementation	OK
Game loop	OK
Player's Data gahtering system	OK
Testing / Gather Data	OK
Data Analysis	OK
Documentation	OK

Table 10.2: Correlation coefficient values

We estimate that we need at least 100 test subjects to perform a suitable analysis of the average player behavior, but with our 30 subject dataset we think it is enough to arrive to a simple equation to represent the player performance, as we mentioned in the beginning. These are the first steps into the creation of a tool that helps the player reach the 'Flow' zone faster.

We succeeded in our application of PCG with DDA techniques to create a first approach of a tool to measure the player performance and the difficulty of a level. In this case we managed to make a formula that makes a numeric representation of how good or how bad the player is reaching the exit of the level, which is the objective that we want them to pursue. This Formula 10.2 represents our final result that we were looking for.

Equation 10.2 Formula to measure the players performance

$$x = \frac{player\_steps - optimal\_path}{opened\_nodes\_bfs - optimal\_path} \tag{10.2}$$

The formula is a normalization of the player performance, were we use the optimal path as minimum and the opened nodes of BFS as the maximum. The formula is used to measure the performance in previous mazes completed by the player. Once the player passes the level, the formula calculates how good or how bad he/she has performed in the maze, placing the player "skills" between the best and the worst result.

With the player performance calculated we now compare it with the previous levels information. We calculate the mean of the previous mazes completed (x) and compare it with the current normalized data (y). If our y is higher than our x we check the last completed level of the player (x_n) and if x_n is less than our y value we decrease our maze size scope for the prediction phase, if not we keep the size. If our current y value is equal to x, then we proceed to compare it with x_n, so if our y value is less than x_n, we keep the size and if y is less or equal than x_n we rise the difficulty in order to create a more challenging maze to the player, if the player has an y value directly higher than x we rise the difficulty taking note that the player is improving constantly.

Finally we enter on our prediction phase in which with our maze size previously assigned

we proceed to create 200 mazes and calculate their difficulty with the next formula:

Equation 10.3 Formula to measure the level of difficulty of a maze

$$x = \frac{opened\_nodes\_bfs - optimal\_path - (maze\_size/2)}{opened\_nodes\_bfs} \tag{10.3}$$

This formula gives us value which we compare with the player performance and search for a maze with a similar value depending if in the previous phase we declare that the player is struggling or search for a higher value if the player is increasing his/her performance.

Chapter 11

Conclusions

With BTP we create a complete maze in which we can establish a size and the algorithm will create a complete dungeon maze with it. We use BFS and A* to measure the difficultly and the performance of the player. With that information we create a formula that represent the player performance and other formula that represent the maze difficulty, with tat information we can increase or decrease the size of the maze.

We accomplish the purpose of the investigation, we found a formula that represent that specific layer of difficultly to the player. The BFS algorithm shows a strong relation to the player's behavior and the data analyzed (player steps) are a good start point to into the the creation of the DDA tool that we want to create in further work.

Even if we succeed in our main goal, we have to mention some problems with the initial plan. Due to the COVID-19 pandemic, we took the choice to travel back to Chile. That trip, plus all the formalities that we had to do when we came back, delayed our work and our time to run tests was consumed into fix bugs and errors of the project. In the end we choose to present our work in September hoping that we could do our defense in better circumstances.

Our results and conclusions were very similar to other projects that uses PCG and DDA to create personalized levels. As they use in the project Polymorph [3], we use a correlation coefficient to find the better connections to the characteristic of the player that we want to measure and drive. But most of the projects that studied the advantages of using a DDA to improve procedural generation systems, they only aim for one approach of PCG or they only use one in specific parts of their games. We aim for a tool that can improve the AI or the rewards of the players as well the complete level. We could manage to create a system that controls the entire game such in Resident Evil 4, but with a complete control over the level design as an initial point.

Chapter 12

Further work

In this chapter we talk about our next steps and various objectives in order to keep investigating in this project.

12.1 Gather data

In order to accomplish our goal of gathering more than 100 test subjects data we will keep searching for players that are willing to play the game and contribute to our investigation. To do so, we are going to participate with our game into more Game Jams in order to make more players around the world give us their data and keep the testing phase in further process. We could add more features to analyze like enemies or rewards to investigate more layers of difficulties into our game.

12.2 Enemies

As we mention in the previous section, one of our next steps in our investigation is to create a simple enemy with specific behaviors that we could track and analyze with the player data. For example, we could create an enemy that chases the player in the maze, we could manage the speed of the enemy, control of its knowledge (if the enemy knows where the player is or not), the quantity of enemies and we could analyze if the player struggles more or less with all that parameters, combining the complete creation of the level and the complete control of the enemies of that level.

12.3 Analyze A*

The results of this investigation shows that the opened nodes (closed list) of BFS are the ones closest to represent the player steps in the maze. This is because the algorithm not use an heuristic that approaches the exit more efficiently, as A*. This works as a compass for the algorithm, so if we give the player a compass in the HUD, we could measure if the player can reach the exit as fast as A*, or if it consolidate our approach with BFS that is a better choice to analyze the player performance.

12.4 Optimization

Many of our systems need a refactor and optimization in order to improve the creation and the data analysis processes, we could make the process of analyzing the player's data automatic

with a proper database or use better structure for the maze creation or path finding algorithms. Those are some of the modules that we could aim to improve in our next iteration.

12.5 Publish a Game

Using our base investigation and the algorithms that we have created, we could create a commercial game and publish it. The game will put to the test our real goal with a specific and achievable result with simple mechanics that create a personalized experience for every player.

Bibliography

[1] Guru99. Incremental model in sdlc: Use, advantage & disadvantage, 2019. URL https://www.guru99.com/what-is-incremental-model-in-sdlc-advantages-disadvantages.html.

[2] Two-Bit History. How much of a genius-level move was using binary space partitioning in doom?, 2019. URL https://twobithistory.org/2019/11/06/doom-bsp.html.

[3] Martin Jennings-Teats, Gillian Smith, and Noah Wardrip-Fruin. *Polymmorph: A Model for Dynamic Level Generation.* Expressive Intelligence Studio, University of California Santa Cruz, 2010.

[4] Amit's Thoughts on Pathfinding. Maze generation algorithm, 2019. URL http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

[5] Jesse Schell. *The Art of Game Design: A book of lenses.* CRC Press, 2008.

[6] Noor Shaker, Antonios Liapis, Julian Togelius, Ricardo Lopes, and Rafael Bidarra. *Procedural Content Generation in Games, Chapter 3 (Constructive generation methods for dungeons and levels).* Springer, 2016.

[7] Gillian Smith. *Game AI Pro 2, Procedural Content Generation An Overview.* CRC Press, 2015.

[8] Wikipedia. Maze generation algorithm, 2019. URL https://en.wikipedia.org/wiki/Maze_generation_algorithm.

[9] Xueqiao (Joe) Xu. Pathfinding.js, 2019. URL https://qiao.github.io/PathFinding.js/visual/.

[10] Mohammad Zohaib. *Dynamic Difficulty Adjustment (DDA) in Computer Games: A Review.* Advances in Human-Computer Interaction, vol. 2018, 2018.

Chapter 13

Appendix

0.1 Randomized Prim's Algorithm

The algorithm starts with a maze or grid full of walls, we randomly select the root node and add its neighbors to a list. while there are nodes in the list we will pick a random node and connect it to one of its neighbor, we do this until all the nodes are visited and connected.

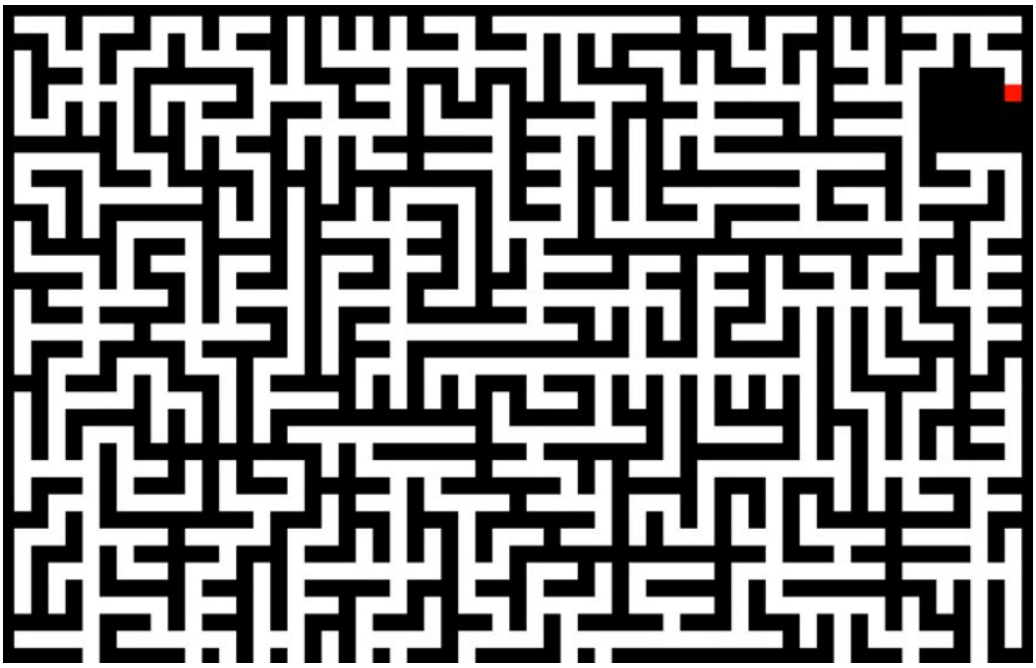


Figure 1: Prim's maze result

0.2 Depth-First Search DFS

Also know as "Recursive Backtracker", this is one of the simplest ways to generate a maze. The algorithm starts in a random node and randomly chooses a cell of it neighbors and continue digging through them until there are all visited once.

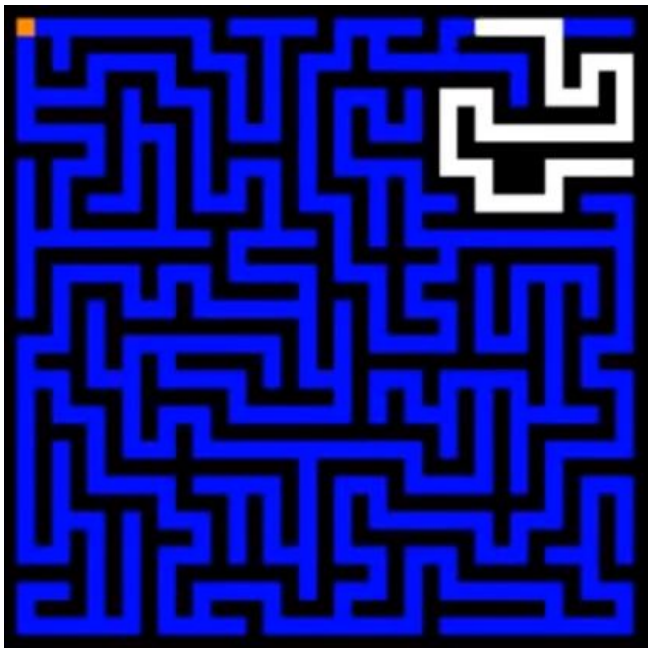


Figure 2: DFS's maze result

0.3 Heuristic

Is any approach to problem solving or self-discovery that employs a practical method that is not guaranteed to be optimal, perfect, or rational, but is nevertheless sufficient for reaching an immediate, short-term goal or approximation. To an A* implementation there are multiple useful heuristics, some of them are: The Manhattan distance, Diagonal distance and Euclidean distance.

Chapter 1

User Manual

1.1 Maze Generator

In order to use the Maze Generator (MG) we need to understand every one of their parameters since we already know how the MG works. The parameter "Maze to reproduce" is used to retrace an already generated maze. "Reproduce int" is used to specify the size of the maze, if this parameter and "Maze to reproduce" are not set then the maze will not be displayed. "Tile size" is the scale of the maze in Game coordinates. "Box height" is the height of the maze walls. The width and height parameters are the size of the maze. The "Tiles" are the "GameObject" pre-established into the process. With "use height tile" and "use cardinals" we specify if we use the 3D version fo the maze and the pathfinding algorithms node constrains of the neighbor gathering in each BFS and A* method.

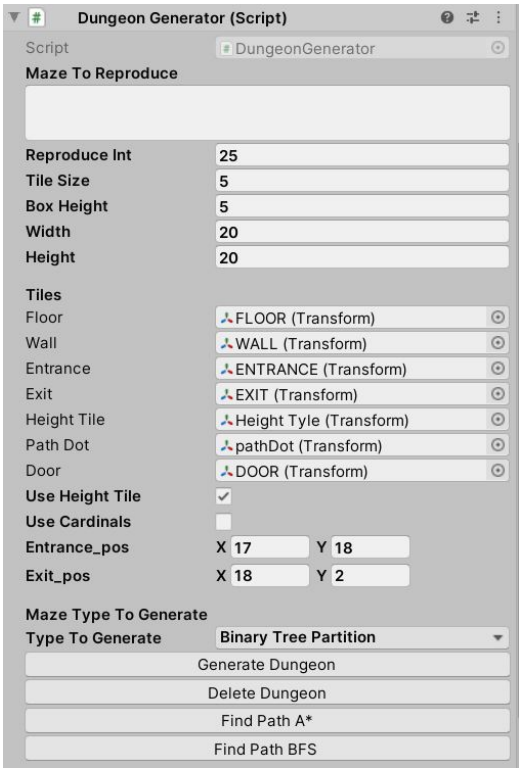


Figure 1.1: Dungeon Generator in Unity3D Editor

The final buttons shows that we can specify the algorithm to be used to generate the maze, generate the maze, delete it or use the pathfinding algorithms.