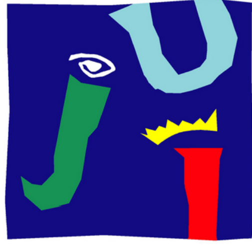


# DAX
# Development and implementation of algorithms with RNG and procedural level generation

**Sergio Juan Pérez Jiménez**

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I

June 28, 2024

Supervised by: Emilio Bueso Aparici.

*To my loved ones*

# Acknowledgments

To begin with, I would like to thank my parents, siblings, girlfriend, and friends, who have always supported me regardless of my decisions, whether they were more successful or, at times, less fortunate. . . .

I also want to thank my tutor, Emilio, because in the moments when I was somewhat lost, he was the one who guided me and helped me materialize the ideas that were initially somewhat scattered.

I also would like to thank Sergio Barrachina Mir and José Vte. Martí Avilés for their inspiring LaTeX template for writing the Final Degree Work report, which I have used as a starting point in writing this report.

# Abstract

Dax is a Rogue-like game in which you step into the shoes of a fox who ventures into the forest in search of his missing wife. It is a 2D pixel-art game with a top-down view, where the level design is procedurally generated.
In this project, various types of procedural generation algorithms have been studied and some features that use RNG have been implemented, ensuring that each play-through is different from the last. Academically, this document consists of the final degree project report for the Video Game Design and Development bachelor's degree at Jaume I University.

**Keywords**

Rogue-like, RNG, procedural generation, Final Degree Work.

# Contents

**Contents** **v**

**1 Introduction** **1**
1.1 Work Motivation . . . . . 1
1.2 Objectives . . . . . 2
1.3 Environment and Initial State . . . . . 2

**2 Planning and resources evaluation** **3**
2.1 Planning . . . . . 3
2.2 Resource Evaluation . . . . . 5

**3 System Analysis and Design** **7**
3.1 Requirement Analysis . . . . . 7
3.2 System Design . . . . . 12
3.3 System Architecture . . . . . 13
3.4 Interface Design . . . . . 13

**4 Work Development and Results** **17**
4.1 Work Development . . . . . 17
4.2 Results . . . . . 24
4.3 Comparison of Procedural Generation Methods for Dungeons . . . . . 25

**5 Conclusions and Future Work** **33**
5.1 Conclusions . . . . . 33
5.2 Future work . . . . . 34

**Bibliography** **35**

**A Source code** **37**

Chapter 1

# Introduction

**Contents**


1.1 Work Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**
1.2 Objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **2**
1.3 Environment and Initial State . . . . . . . . . . . . . . . . . . . . . . . . **2**


I knew that I wanted to explore procedural generation and implement it in a game. However, I wasn't sure which genre to choose because when I think of procedural generation, the first genres that come to mind are survival or sandbox games. After all, Minecraft has been a significant influence since its release, so it's natural for it to be the first thing that comes to mind. I also recalled the Diablo series, which, despite having a pre-designed map, features procedurally generated dungeons.

Ultimately, since the genre I've engaged with the most this year is Rogue-lites, specifically Cult of the Lamb, Moonlighter, and Hades, I decided to focus on procedural dungeon generation for a Rogue-like game, more in the style of The Binding of Isaac than the aforementioned three. This decision wasn't because I prefer The Binding of Isaac, but because I appreciate its approach to managing dungeons and in-game progression. In comparison, Hades has significant in-game progression but lacks the procedural generation I was interested in, while Cult of the Lamb and Moonlighter have most of their progression in the meta-game.

## 1.1 Work Motivation

The motivation for creating Dax stems from my desire to combine procedural room-based level generation with a system capable of integrating prefabricated rooms alongside "default" rooms. This approach allows for the implementation of "special" rooms without

any additional effort once the procedural generation is established. Additionally, the Rogue-like genre offers a high level of replayability, and if well-balanced, provides great satisfaction as players learn enemy movements and progressively advance further.

Beyond wanting to explore different methods of procedural generation, which has always intrigued me, I aimed to investigate the mechanisms for randomizing various aspects of a Rogue-like game. This includes enemies, items, and considering multiple factors to control the randomization's effectively.

## 1.2 Objectives

The primary objective of the Dax project is to develop a 2D Rogue-like video game with a top-down perspective, featuring procedurally generated levels to ensure each game session is unique. The specific objectives include:

- **Procedural Generation:** Implement an efficient procedural generation algorithm created for loading any kind of room of the dungeon without trouble.
- **Engaging Gameplay Mechanics:** Design gameplay mechanics that make the game attractive for players, with a player dynamic movement and challenging enemies.
- **High Replayability:** Ensure each playthrough is different to enhance replayability.
- **Academic Contribution:** Document the development process, challenges, and solutions in a detailed report as part of the final degree project for the Video Game Design and Development bachelor's degree at Jaume I University.

This project aims to contribute significantly to the academic field of game design and development while providing a polished and enjoyable gaming experience.

## 1.3 Environment and Initial State

The development of Dax was carried out using Unity for game development, with C# for scripting, and Libresprite for pixel-art designing. Unity has a great potential in 2D video games, and Libresprite is a nice free pixel-art software. Some sprites were obtained from the Unity Asset Store and Itch.io.

The initial state of the project involved creating a basic framework for procedural generation and defining the main gameplay mechanics. This includes classes like Player.cs, Crawler.cs, CrawlerController.cs, DungeonCreator.cs, Room.cs and RoomController.cs. There was no enemies, no bosses, no special rooms.

CHAPTER 2

# Planning and resources evaluation

**Contents**


2.1 Planning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.2 Resource Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5


This chapter is the most technical part of the work, and shows all the work assessed from objective information and its estimated cost.

## 2.1 Planning

At first, the initial idea of how my project would turn out differed slightly from the final result, not because of an inability to achieve the outcome, but rather due to a shift in focus towards creating a more enjoyable game that reflects my personal gaming style. Initially, the project was intended as a study of procedural algorithms; however, a video game must be entertaining to make for understanding the algoriths.

The estimated hours and tasks at the beginning of development can be seen in the following table (see Table 2.1).

To demonstrate the actual process through which the game state has evolved, a Gantt chart has been added. This chart not only displays the hours required for each part of the development but also shows the sequence and time during which tasks were carried out, even when multiple tasks were being executed concurrently. This approach was necessary as certain elements of the game were interdependent and could only be implemented once others were completed (see Figure 2.1). In addition to the game development itself, approximately fifty additional hours were dedicated to game documentation and algorithm research.

Figure 2.1: Gantt chart (made with TeamGantt)

| **Task Name** | **Estimated Hours** |
|---|---|
| Game Mechanics Implementation | 50 hours |
| Procedural Level Generation | 50 hours |
| Character and Enemy AI | 35 hours |
| Boss Battles Implementation | 40 hours |
| UI/UX Design | 20 hours |
| Pixel Art Assets Creation | 50 hours |
| Bug Fixing and Testing | 35 hours |
| Documentation and Reports | 20 hours |
| **Total** | **300 hours** |

Table 2.1: Estimated hours for each task

## 2.2 Resource Evaluation

Taking into account the hours required to carry out the game and adding the tasks that were initially replaced by freely available assets, the following is an estimation in hours and costs necessary to complete the project.

### 2.2.1 Human Resources

The human resources involved in the project primarily include the development team, composed of roles such as project manager, programmer, level designer, translator, audio designers and UI/UX designer. The estimated time allocation for each major task is:

- Programmer: Estimated at 230 hours
- Art Designer: Estimated at 50 hours
- UI/UX Designer: Estimated at 20 hours
- Audio Designer: Estimated at 25 hours
- Translators: Estimated at 40 hours
- QA Tester: Estimated at 15 hours

### 2.2.2 Equipment

The equipment necessary for the development of *Dax* as a full game includes the following:

- **Hardware:**
  - Development PCs with sufficient specifications to run Unity and other required software.

- Peripherals such as keyboards, mice, sound systems and monitors.

- **Software:**
  - Unity for game development.
  - Visual Studio for coding.
  - Libresprite for creating the assets.
  - Cloud storage devices for backups and version control, as Github.
  - Git for version control.
  - Audacity for audio design.
  - TeamGantt for project management and planning.
  - Microsoft Office Excel for the localization.

### 2.2.3 Cost Estimation

The estimated cost of resources includes both the human labor and the necessary equipment, without using any public assets library is:

- **Human Resources:**
  - Programmer: Estimated at 10,77€/hour
  - Art Designer: Estimated at 10,56€/hour
  - UI/UX Designer: Estimated at 15,38€/hour
  - Audio Designer: Estimated at 7,29€/hour
  - Translators: Estimated at 12,03€/hour
  - QA Tester: Estimated at 14,10€/hour
- **Hardware:**
  - Development PCs: 5,000€
  - Peripherals: 500€
- **Software Licenses:**
  - Unity Pro License: 185€/month
  - Visual Studio License: 45€/month
  - Libresprite (Free)
  - Git (Free, unless using private repositories)
  - TeamGantt License: 50€/month

Chapter 3

# System Analysis and Design

Contents

3.1 Requirement Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7**
3.2 System Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **12**
3.3 System Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13**
3.4 Interface Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **13**

This chapter delves into the detailed analysis and design of the system. All aspects of the system's requirements, both functional and non-functional, are explored, providing a comprehensive blueprint for the development process.

## 3.1 Requirement Analysis

To carry out a job, it is necessary to perform a preliminary analysis of its requirements. In this section, the functional and non-functional requirements of the presented work are detailed.

### 3.1.1 Functional Requirements

A functional requirement defines a function of the system that is going to be developed. This function is described as a set of inputs, its behavior, and its outputs[6]. The functional requirements for this project are as follows:

- **R1: Procedural Generation of Levels**

| Input: | Enter any floor |
| --- | --- |
| Output: | A procedurally generated dungeon with a mix of prefabricated and default rooms |
| The system will procedurally generate levels by creating dungeons composed of rooms that are a mix of prefabricated special rooms and procedurally generated default rooms. | |

Table 3.1: Functional requirement «Procedural Generation of Levels»

- **R2: Player Character Actions**

| Input: | Player input from keyboard/controller |
| --- | --- |
| Output: | Corresponding player actions (move, attack, dash, etc.) |
| The system will process player inputs to control the character's movements, attacks, dashes, and other actions. | |

Table 3.2: Functional requirement «Player Character Actions»

- **R3: Enemy AI and Behavior**

| Input: | Game state and player position |
| --- | --- |
| Output: | Enemy actions and reactions |
| The system will manage enemy AI to react to the player's position and actions, including movement, attack patterns, and special behaviors. | |

Table 3.3: Functional requirement «Enemy AI and Behavior»

- **R4: Boss Fights**

| Input: | Enter the Boss room |
| --- | --- |
| Output: | Boss battle sequence |
| The system will initiate and manage boss fights, with unique attack patterns and behaviors for each boss. | |

Table 3.4: Functional requirement «Boss Fights»

- **R5: User Interface**

| Input: | Game events and player interaction |
|---|---|
| Output: | Display of game menus, HUD, and other UI elements |
| The system will provide a user interface to display game information, including menus, HUD, player stats, and minimap. | |

Table 3.5: Functional requirement «User Interface»

- **R6: Item Collection and Management**

| Input: | Player interaction with chests |
|---|---|
| Output: | Item effects are applied |
| The system will handle item collection, and usage, applying effects to the player character as appropriate. | |

Table 3.6: Functional requirement «Item Collection and Management»

- **R7: Pause**

| Input: | Player presses pause key/button |
|---|---|
| Output: | The game is paused and a *Pause Menu* is displayed. |
| The system will handle the time multiplier, pausing the game, and the *Pause Menu* will be displayed. | |

Table 3.7: Functional requirement «Pause»

- **R8: Quit Run**

| Input: | Player clicks the button *Exit* in the *Pause Menu* |
| --- | --- |
| Output: | The game returns to the *Main Menu*. |
| The scene changes to the menu, saving the last floor achieved if it is greater than it was before the run. | |

Table 3.8: Functional requirement «Quit Run»

- **R9: Resume Game**

| Input: | Player clicks the button *Resume* in the *Pause Menu* or presses the pause input. |
| --- | --- |
| Output: | The game is resumed. |
| The system will handle the time multiplier, resuming the game, and the *Pause Menu* will be hidden. | |

Table 3.9: Functional requirement «Resume Game»

- **R10: Open Settings**

| Input: | Player clicks the button *Settings* in the *Pause Menu* or *Main Menu*. |
| --- | --- |
| Output: | The settings tab is opened. |
| The system will handle the menu tabs, hiding the unnecessary buttons and showing the *Settings* tab. | |

Table 3.10: Functional requirement «Open Settings»

- **R11: Change volume**

| | |
|---|---|
| Input: | Player uses the volume sliders. |
| Output: | The music / sfx volume changes. |
| The system will handle the audio manager, adapting the volume of the music / sfx with a logarithmic function. | |

Table 3.11: Functional requirement «Change volume»

- **R12: Quit game**

| | |
|---|---|
| Input: | Player clicks the button *X* in the *Main Menu*. |
| Output: | The game is closed. |
| The game is closed. | |

Table 3.12: Functional requirement «Quit game»

- **R13: Room closing**

| | |
|---|---|
| Input: | The player enters a room with enemies. |
| Output: | The room is closed. |
| The doors in the room are closed when the player enters the room, making it impossible to leave the room until every enemy is beaten. | |

Table 3.13: Functional requirement «Room closing»

- **R14: Play**

| | |
|---|---|
| Input: | Player clicks the *Play* button. |
| Output: | The game begins. |
| The dungeon is generated and the game starts. | |

Table 3.14: Functional requirement «Play»

### 3.1.2 Non-functional Requirements

Non-functional requirements impose conditions on the design or implementation to meet performance, safety, or reliability constraints[8]. The non-functional requirements for this project include:

- **R15: Performance**
  - *Description:* The system will perform efficiently with minimal delay, ensuring smooth gameplay even during intensive boss battles or when multiple enemies are on screen. The floor loads should not take long.
- **R16: Usability**
  - *Description:* The game will have an intuitive user interface and controls, making it accessible and easy to learn for new players.
- **R17: Compatibility**
  - *Description:* The system will be compatible with Windows 7 or higher and MacOS X 10 or higher.
- **R18: Reliability**
  - *Description:* The system will be reliable, with none bugs or crashes, providing a stable gaming experience.
- **R19: Save Data**
  - *Description:* The system will save correctly, both the player settings and the last floor reached, even when closing the game.

## 3.2 System Design

This section presents the operational design of the system to be carried out:

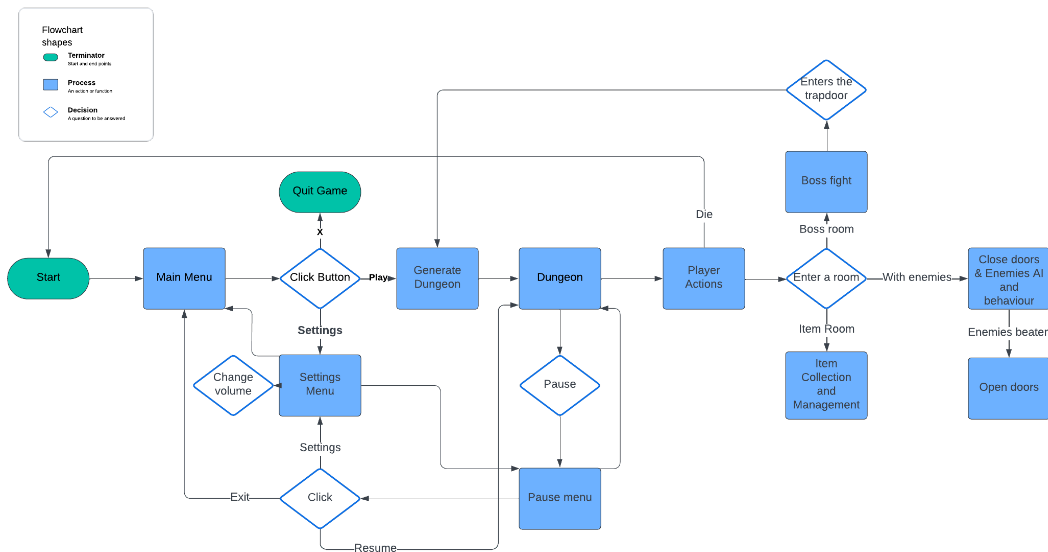



Figure 3.1: Flowchart (made with Lucid)

## 3.3 System Architecture

- Minimum requirements [1]
  - 105 Mb of free space.
  - 4 Gb RAM.
  - Windows 7 o greater o MacOS X 10.8 or greater.
  - DX9 o DX11.
- Recomended requirements
  - 105 Mb of free space.
  - 8 Gb RAM.
  - Windows 7 or greater o MacOS X 10.8 or greater.
  - Graphic card INTEL GRAPHIC 4000 or greater, or Nvidia o ATI with 1G VRAM or greater.

## 3.4 Interface Design

The main menu interface is pretty simple (see Figure 3.2), with a button for starting the game, other for the settings menu (see Figure 3.3 and Figure 3.4) and the last one for quitting the game . Same happens with the pause menu (see Figure 3.5), having,

instead of a *Starting Game* button, a *Resume* button, and instead of a *Quit* button, it exists the run. About the GUI, it is composed by a health system made of hearts, each one represents four *HP*, situated in the top-left corner of the screen, a mini-map that saves the visited rooms and a stats overview (see Figure 3.6).

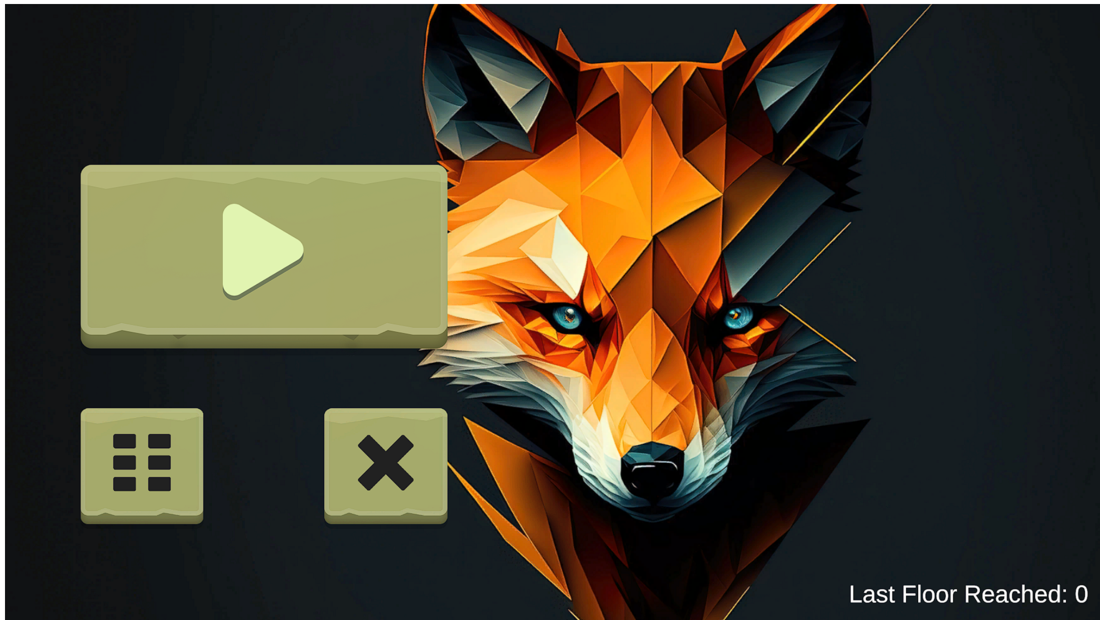



Figure 3.2: Main Menu

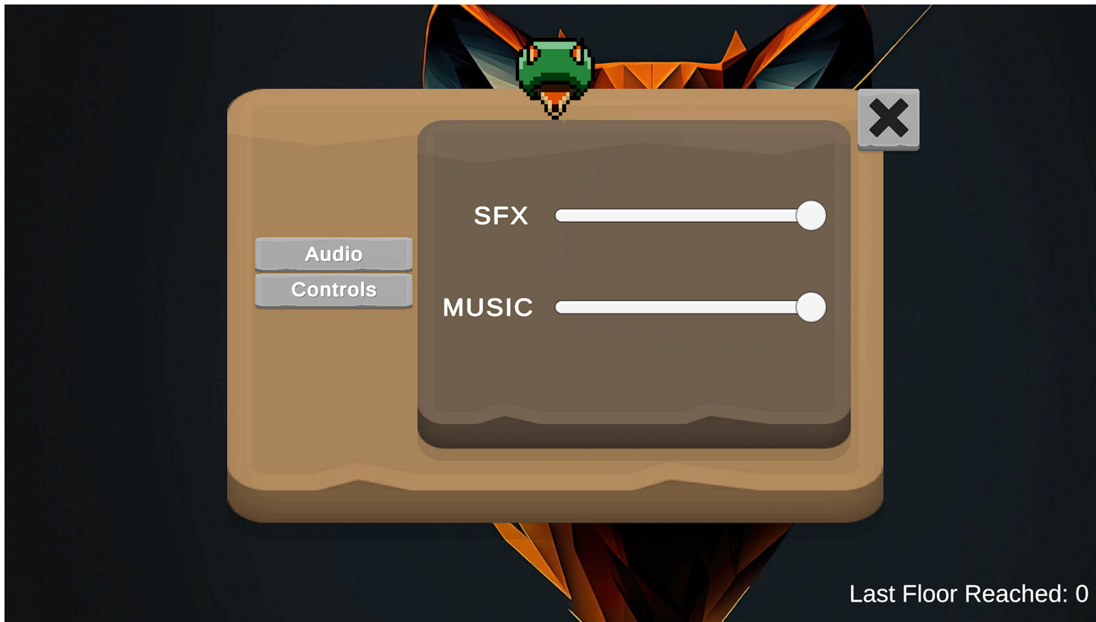



Figure 3.3: Audio Settings

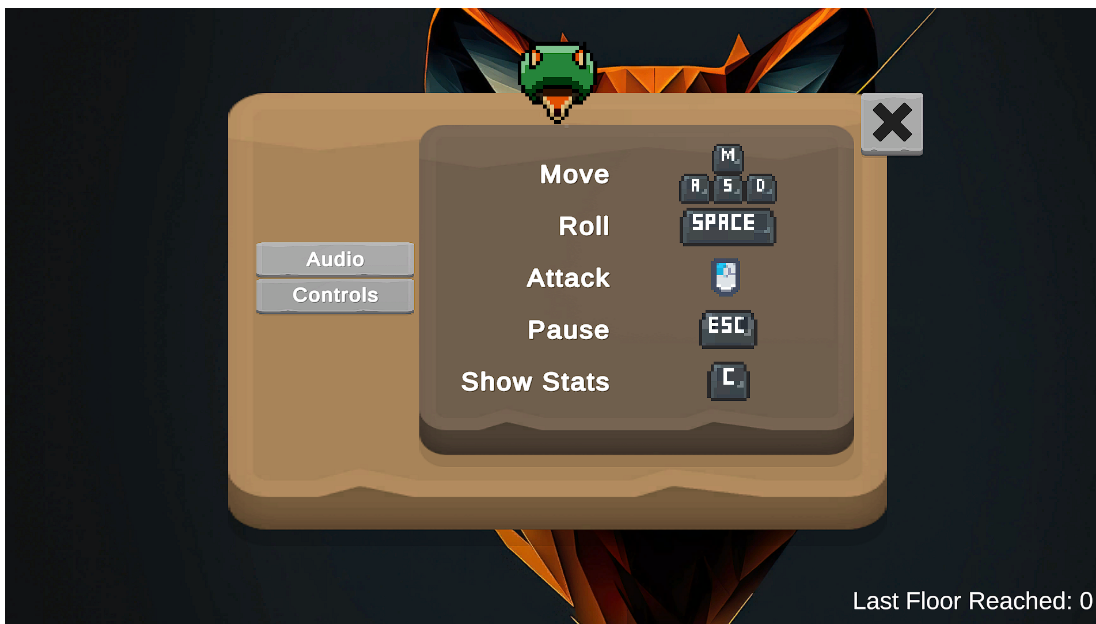



Figure 3.4: Controls Setting

Figure 3.5: Pause Menu

Figure 3.6: GUI

CHAPTER 4

# Work Development and Results

**Contents**

4.1 Work Development . . . . . . . . . . . . . . . . . . . . . . . . . . . . **17**
4.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **24**
4.3 Comparison of Procedural Generation Methods for Dungeons . . . . . **25**

The developed work and the obtained results should be made explicit in this chapter. All possible deviations from the initial planning should be detailed and justified. In this way, readers of the memory must be able to understand the possible reasons for discrepancies between the objectives of the work, the planning that supposedly allowed to obtain it, and the final results achieved.

## 4.1 Work Development

In this section, the most relevant aspects of the developed work will be explained in detail. The structure follows a chronological order, highlighting significant milestones and issues encountered during the project.

### 4.1.1 Initial Planning and Objectives

The primary objective of this project was to develop a Rogue-like game featuring procedurally generated levels. The project was structured around several key components: Procedural Generation of Levels, Player Character Actions, Enemy AI and Behavior, Boss Fights, User Interface (UI), Item Collection and Management, Game Mechanics, and Testing and Bug Fixing.

### 4.1.2 Development Phases

#### Procedural Generation of Levels

The first phase focused on creating a system for procedural generation of dungeon levels. This system aimed to provide players with a unique experience every time they played the game by generating dungeons composed of both prefabricated special rooms and procedurally generated default rooms. The algorithm used for this task was a combination of Depth-First Search (DFS) and random selection.

The DFS algorithm was chosen for its efficiency in creating connected paths, ensuring that all rooms in the dungeon were accessible. Random selection was used to introduce variability in the room types and layouts, enhancing replayability. The primary challenge in this phase was balancing the mix of room types to maintain a coherent and engaging dungeon layout.

Specifically, this depth-first search iterates simultaneously from the same starting point (the origin coordinates) and moves randomly, taking a random number within a stipulated range as the limit. Each entity that iterates by adding to a list of "visited" points is called a "Crawler"[3].

#### Player Character Actions

The second phase focused on implementing responsive and intuitive controls for the player character. This included handling inputs for movement, attacks, and dash, using the New Input System (see Figure 4.1 and Figure 4.2). The last two actions have a cool-down, in order to balance the game, so the player cannot attack or dash infinitely.

Animations are synchronized with player actions to provide visual feedback and enhance immersion.

Overall, the player control system was successfully integrated.

Figure 4.1: Player attacking.

Figure 4.2: Player dashing.

### Enemy AI and Behavior

Developing the AI for enemies was one of the most challenging aspects of the project. The AI needed to react dynamically to the player's actions and position. This involved programming different behaviors for various enemy types, including movement patterns, and attack strategies (see Figure 4.3).

Figure 4.3: Enemies attacking.

The AI system used state machines to manage different behaviors and transitions between them based on game state and player actions.

The enemy generation is prepared for generating the random number of enemies without colliding with any object or enemy. This was implemented by creating a grid, with the same size as the tile-set. When a room is created, the *Enemy Spawner* chooses random positions in the grid, checking if they are "occupied", and generating the enemies correctly (see Figure 4.4 and Figure 4.5).

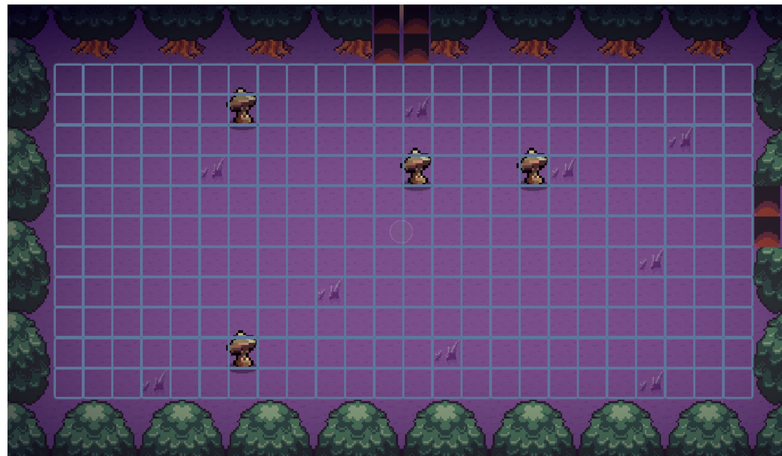


Figure 4.4: Tiles without obstacles

Figure 4.5: Tiles with obstacles

### Boss Fights

Boss fights are a critical component of any Rogue-like game, providing significant challenges. Each boss in the game was designed with unique attack patterns and behaviors, requiring players to develop different strategies to defeat them.

For example, the Slime boss attacks by jumping at you. Its attack has a lot of cooldown, so the player can attack it while it is not jumping, but carefully, because once it jumps, it does quite fast (see Figure 4.6).

Figure 4.6: Slime Jumping.

### User Interface (UI)

An intuitive and informative UI is essential for player interaction and immersion. The UI needed to display critical game information, including health, stats, and a mini-map, in a clear and accessible manner.

The design process involved several iterations to ensure that the UI was both functional and visually appealing. The health system uses images that are placed but inactive at the beginning, so there is a limit the health can scale, the limit is 120hp, because there are only thirty hearts that fits in the screen without overlapping the mini-map or the doors (see Figure 4.7 and Figure 4.8). The stats are in the bottom-left corner, so they are accessible but not annoying for the gameplay (see Figure 4.7).

Figure 4.7: Hearts

Figure 4.8: Maximum health.

Figure 4.9: Stats.

The mini-map is implemented using a camera aiming in the y-axis (see Figure 4.10). The image of that camera is saved in a texture, so applying that texture to an image makes visible the mini-map objects. Every time a room is registered, its mini-map game object is created, instantiated and deactivated. When the player visits a room, the mini-map room is activated, so it is visible in the screen.

Figure 4.10: Minimap.

#### Item Collection and Management

Item collection and management added depth to the gameplay, allowing players to collect items they found in the dungeon.

Each item provides some different stats, and every time the player opens a chest, a random item is collected (see Figure 4.11).

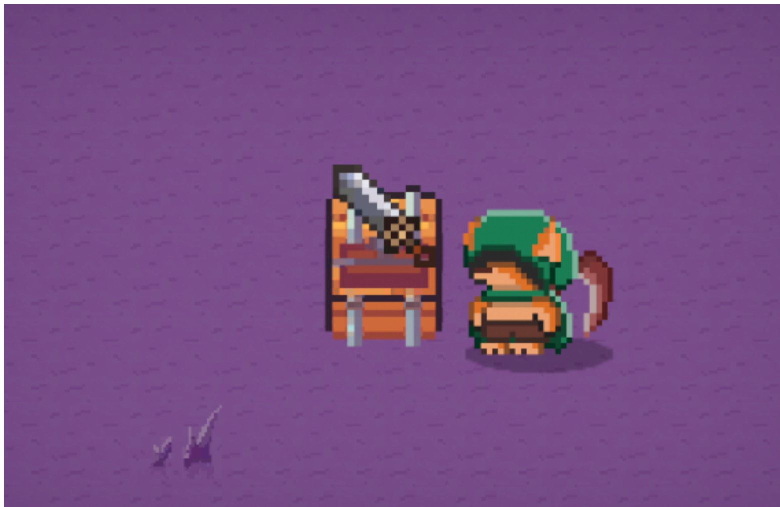


Figure 4.11: Item collection

### 4.1.3 Deviations from Initial Planning

During the development process, some deviations from the initial plan occurred due to unforeseen challenges. The complexity of AI development took longer than expected,

requiring additional time for iterative testing and adjustments. Additionally, extra time was needed for optimization and bug fixing to achieve the desired performance and stability. Initially, the special dungeon rooms were implemented by a complex method, that was executed just after the dungeon was completely generated, deleting the room that was in the needed position, but this method was deleted, so now there are only two methods that calculate the position where the special rooms have to be placed so the rooms are placed in once:

#### Item Room Position

The method generates a list of item room locations from a given collection of room positions (Vector2Int). It converts the collection to a list and checks if it contains any rooms. If not, it returns null. It then determines the number of item rooms to generate, based on a random fraction of the total room count. The method iterates, selecting random rooms from the list while ensuring they are not the origin room, not adjacent to the origin, and not at the maximum distance. Selected rooms are added to the "generatedItemRooms" list and removed from the original list to prevent duplicates. The method returns the list of generated item room locations.

#### Boss Room Position

As the Boss Room is the farthest one from the origin (0, 0), the method finds the room farthest from the origin in a list of room positions represented by Vector2Int. It initializes the farthest room to the first room in the list and calculates its distance from the origin. It then iterates through the list, updating the farthest room and maximum distance whenever a room farther from the origin is found. Finally, it returns the position of the farthest room.

## 4.2 Results

The results of the project are described below, referenced to the initial objectives.

### 4.2.1 Achieved Milestones

The project successfully achieved several key milestones:

- Procedural Level Generation: The procedural generation system was successfully implemented, providing varied and engaging dungeon layouts for each playthrough.
- Player actions: Responsive and intuitive player controls were developed, enhancing the overall gameplay experience.
- Enemy AI and Behavior: A dynamic and challenging enemy AI system was created, providing engaging combat encounters.

- Boss Fights: Unique and challenging boss fights were designed and implemented, adding significant depth to the gameplay.
- User Interface: An intuitive and informative UI was developed, clearly displaying essential game information.
- Item Collection and Management: An efficient item collection and management system was implemented, adding strategic depth to the game.

### 4.2.2 Applications and Future Releases

The developed game has several potential applications and future release plans:

- Educational Tool: The game can be used as an educational tool to demonstrate procedural generation techniques and AI behavior.
- Open Source Release: The project code will be made available on GitHub, allowing other developers to study and contribute to the project.

## 4.3 Comparison of Procedural Generation Methods for Dungeons

Procedural generation is a technique widely used in game development to create content algorithmically rather than manually. This method offers a variety of advantages such as reducing the amount of hand-crafted content required and increasing the replayability of the game by providing a unique experience each time. In this section, we will compare the procedural generation method employed in our project, which utilizes simultaneous depth-first search with random movement (crawlers), against two other widely used methods: cellular-automata and marching-squares.

### 4.3.1 Crawler-Based Procedural Generation

Our approach to dungeon generation uses a crawler-based depth-first search algorithm. This method involves multiple entities called "crawlers" that start from the same origin point and move randomly within a predetermined range. Each crawler iterates through the dungeon space, adding points to a list of "visited" locations and creating paths and rooms in the process (see Figure 4.12).

#### Advantages

- Simplicity and Flexibility: The crawler-based method is relatively simple to implement and highly flexible. Crawlers can be programmed to follow different rules, such as favoring certain directions or avoiding previously visited areas, which can create a wide variety of dungeon layouts.

Figure 4.12: Crawler-Based Procedural Generation [4]

- Natural Layouts: This method often produces layouts that feel organic and natural. The random movement of crawlers tends to create winding paths and irregular room shapes, which can be more immersive for players compared to more structured methods.
- Scalability: The crawler method scales well with complexity. By increasing the number of crawlers or adjusting their movement patterns, developers can easily control the density and complexity of the generated dungeon.
- Dynamic Room Creation: Crawlers can be programmed to create different types of rooms (e.g., special rooms, boss rooms) based on specific conditions. This allows for dynamic and varied dungeon experiences.
- Performance: This approach is typically efficient in terms of memory usage because it operates directly on a graph or grid structure without needing to maintain a large amount of additional data. It can be effective for generating complex interconnected structures like dungeons or cave systems if they are not extremely large. Its cost is $O(C \times I)$, where:
  - C: Number of crawlers used.
  - I: Number of iterations each crawler performs.

**Disadvantages**

- Control and Predictability: While the randomness can be a strength, it can also lead to a lack of control over the final layout. Ensuring that all rooms are accessible and that the dungeon has a coherent structure can be challenging.

- Balance: Balancing the difficulty and ensuring that items and enemies are distributed fairly throughout the dungeon can be difficult due to the random nature of the method.

### 4.3.2 Cellular Automata-Based Procedural Generation

Cellular automata is another popular method for procedural dungeon generation. This technique uses a grid of cells that evolve according to a set of rules. Each cell can be in one of several states (e.g., wall, floor), and its state is determined by the states of its neighboring cells[5](see Figure 4.13).

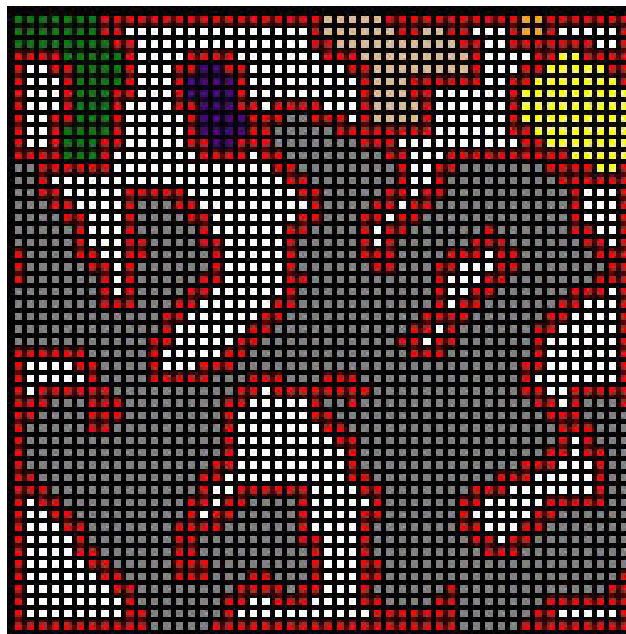


Figure 4.13: Cellular Automata-Based Procedural Generation [9]

#### Advantages

- Control and Structure: Cellular automata allow for greater control over the final layout. By carefully tuning the rules that govern cell evolution, developers can create dungeons with specific properties, such as a certain level of connectivity or room size.

- Balance: The structured nature of cellular automata makes it easier to balance the distribution of rooms, enemies, and items. Developers can impose constraints to ensure that dungeons meet certain criteria.

- Procedural Refinement: Cellular automata can be used in multiple passes to refine the dungeon layout. For example, an initial pass might create the basic layout, while subsequent passes can add details such as corridors and special rooms.

**Disadvantages**

- Complexity: Cellular automata can be more complex to implement and require careful tuning of rules to achieve the desired results. The initial setup and fine-tuning process can be time-consuming.
- Cost: This method can be more efficient than crawler-based methods if both are used on fixed grids, as it typically involves fewer iterations, but it is still so computationally intensive. Its cost is O(W×H×I), where:
  - W: Width of the map in cells
  - H: Height of the map in cells.
  - R: Resolution of the noise generation (detail frequency).
- Uniformity: While cellular automata can create structured and balanced dungeons, they may lack the natural, organic feel that crawler-based methods provide. The resulting layouts can sometimes feel too regular and predictable.
- Less Dynamic Room Creation: Adding special rooms or unique features dynamically can be more challenging with cellular automata compared to the flexible crawler approach.

### 4.3.3 Marching Squares-Based Procedural Generation

The marching squares algorithm is another method often used for generating organic and natural-looking dungeons. This method works by dividing the dungeon space into a grid and then determining the walls and floors by evaluating the corners of each grid square[7] (see Figure 4.14).

**Advantages**

- Natural and Organic Layouts: The marching squares method excels at producing layouts that mimic natural formations like caves and caverns. This creates an immersive environment that can enhance the player's experience of exploration.
- Smooth Transitions: Unlike grid-based methods, marching squares can create smooth transitions between different areas, resulting in more visually appealing and less blocky dungeon structures.
- Control over Density: Developers can adjust the density of the dungeon by altering the threshold values used in the algorithm. This allows for easy tuning of how open or cramped the dungeon feels.

Figure 4.14: Marching Squares-Based Procedural Generation [2]

- Flexibility in Design: By combining marching squares with other techniques, such as Perlin noise, developers can create complex and varied dungeon layouts that offer a unique blend of procedural and hand-crafted elements.

**Disadvantages**

- Complex Implementation: The marching squares algorithm can be more complex to implement compared to simpler methods like crawlers or cellular automata. It requires a good understanding of the underlying mathematics and algorithms.

- Performance Overhead: Generating smooth and organic layouts can be computationally expensive, especially for larger dungeons. Optimization techniques are necessary to maintain performance. Its cost is $O(W \times H \times R)$, where:

  - W: Width of the map in cells
  - H: Height of the map in cells.
  - R: Resolution of the noise generation (detail frequency).

- Difficulty in Adding Special Rooms: While marching squares can create natural layouts, adding specific types of rooms (e.g., boss rooms, treasure rooms) can be challenging. It often requires additional steps or hybrid approaches to integrate these features seamlessly.

### 4.3.4 Case Study: Applying All Three Methods

To illustrate the differences between these methods, consider a dungeon crawler game where the goal is to explore a series of interconnected rooms, fight enemies, and find treasure.

#### Crawler-Based Generation

In our project, the crawler-based method starts with multiple crawlers at the dungeon entrance. Each crawler moves randomly, carving out paths and rooms as it goes. Special conditions are set for creating specific types of rooms, such as treasure rooms or boss rooms. This method results in a dungeon that feels labyrinthine and unpredictable, enhancing the sense of exploration.

#### Cellular Automata-Based Generation

Using cellular automata, the dungeon is initially set up as a grid of cells, all in a "wall" state. A set of rules is applied iteratively to convert some of these cells into "floor" states, forming rooms and corridors. For instance, a common rule might be: "If a cell has three or more neighboring floor cells, it becomes a floor cell." This rule is applied multiple times to generate the basic layout.

Additional rules can be introduced to ensure connectivity and balance. For example, once the basic layout is generated, another pass might add corridors to connect isolated rooms, ensuring that the player can reach all areas of the dungeon.

#### Marching Squares-Based Generation

Using the marching squares algorithm, the dungeon space is divided into a grid, and the algorithm evaluates the corners of each grid square to determine wall and floor placement. This method can produce a layout that feels like a natural cave system with smooth transitions between areas. For instance, applying Perlin noise to the grid values before running the marching squares algorithm can result in a dungeon with varying room sizes and shapes, enhancing the organic feel of the environment.

### 4.3.5 Conclusion

All three methods, crawler-based, cellular automata, and marching squares have their strengths and weaknesses. The choice of method depends on the specific requirements of the game and the desired player experience.

The crawler-based method excels in creating natural, unpredictable layouts that enhance the sense of exploration but can be harder to control and optimize. Cellular automata provide more control and efficiency, producing structured, balanced dungeons but may lack the organic feel of crawler-generated layouts. Marching squares offer a highly natural and organic dungeon layout, which can greatly enhance immersion, but may be more complex to implement and optimize.

In this project, the crawler-based method was chosen for its ability to create dynamic, varied dungeons that contribute to the replayability of the game and the capacity of implement prefabricated rooms. By carefully tuning the behavior of crawlers and incorporating special room conditions, we achieved a balance between randomness and structure, resulting in a compelling dungeon-crawling experience. However, exploring hybrid approaches that combine elements of cellular-automata and marching-squares could provide even greater flexibility and variety in dungeon design.

| Aspect | Depth-First Search Crawler-Based Generation | Cellular Automata | Organic Noise-Based Generation |
|---|---|---|---|
| **Pros** | • High customization with various crawler behaviors and constraints.<br>• High control over dungeon layout and room placement.<br>• Efficient for small to medium-sized dungeons. | • Easy to implement with simple, repeatable rules.<br>• Natural cave-like structures.<br>• Moderate performance for simple caves. | • Highly realistic natural landscapes.<br>• High flexibility in shaping and detailing.<br>• Suitable for diverse environments. |
| **Cons** | • Requires careful management of crawler interactions.<br>• Medium complexity in implementation.<br>• Can be less natural and random. | • Can become computationally intensive with extensive grids.<br>• Limited control over specific layout.<br>• Sometimes repetitive patterns. | • Computationally intensive.<br>• Complex implementation.<br>• Requires optimization for large-scale use. |

Table 4.1: Comparison of Procedural Generation Algorithms

CHAPTER 5

# Conclusions and Future Work

**Contents**

5.1 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **33**
5.2 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **34**

In this chapter, the conclusions of the work, as well as its future extensions are shown.

## 5.1 Conclusions

The completion of this project has been a significant learning experience that has enriched both my technical skills and understanding of procedural content generation in game development. Implementing the crawler-based procedural generation method required a deep dive into algorithm design and programming, offering insights into the complexities and trade-offs involved.

Professionally, this project has equipped me with practical skills in software development and problem-solving. It underscored the importance of iterative development cycles and the need for adaptability when faced with challenges. It also demonstrated the relevance of theoretical concepts from courses such as algorithms, data structures, and artificial intelligence in real-world applications.

On a personal level, managing this project taught me valuable lessons in time management, perseverance, and collaboration. Balancing project demands with other academic and personal commitments reinforced the importance of effective planning and communication.

The project's connection to my coursework was evident throughout, as it integrated concepts and techniques from various subjects directly applicable to game development.

This practical application solidified my understanding and underscored the interdisciplinary nature of effective software engineering.

In conclusion, this project has been a pivotal experience that deepened my technical expertise, enhanced my problem-solving skills, and provided a solid foundation for future projects in game development and procedural content generation. It has been a journey of learning and growth, preparing me well for challenges and opportunities in my future career.

## 5.2 Future work

Firstly, while the crawler-based procedural generation method has proven effective, there is potential to explore hybrid approaches. Combining the strengths of crawler-based methods with those of cellular automata and marching squares could result in even more varied and engaging dungeon layouts. For instance, integrating cellular automata to refine room connectivity or using marching squares for creating organic cave-like structures could provide additional diversity and complexity to the generated dungeons.

Secondly, optimization is a critical area for future work. Improving the performance of the procedural generation algorithm, particularly for larger and more complex dungeons, would enhance the overall user experience. Techniques such as spatial partitioning, multi-threading, and efficient memory management could be investigated to achieve this goal.

Expanding the variety of special rooms and events within the dungeon is another potential area for development. Introducing more diverse room types, environmental hazards, puzzles, and interactive elements could enrich the gameplay experience. Additionally, procedural narrative elements could be integrated to create a more immersive and engaging story within the dungeon.

Personally, I am committed to continuing the development of this project. The knowledge and skills I have gained through this experience have motivated me to further refine and expand the game. I plan to implement some of the suggested improvements and explore new features that can enhance the overall quality and appeal of the game.

# Bibliography


[1] System requirements for unity 2022.3. https://docs.unity3d.com/Manual/system-requirements.html.

[2] Jasper Flick. Marching squares. https://catlikecoding.com/unity/tutorials/marching-squares/.

[3] JVCOB. Binding of isaac clone - unity beginner tutorial series. https://www.youtube.com/watch?v=jQClYQ4cK-Elist=PLosGp2abdYXQF3ukYDoB3mzX0h0eKWXkQ.

[4] Maurits Laanbroek. Procedural dungeon generation for houdini and unity. https://www.youtube.com/watch?v=uLiHWJP-GBg.

[5] Wikipedia. Cellular automaton. https://en.wikipedia.org/wiki/Cellular$_{a}utomaton$.

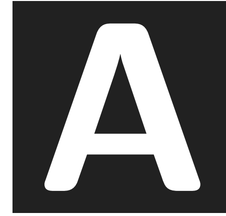
[6] Wikipedia. Functional requirements. http://en.wikipedia.org/wiki/Functional_requirements. Accessed: 2019-02-28.

[7] Wikipedia. Marching squares. https://en.wikipedia.org/wiki/Marching$_{s}quares$ : : $text = In$

[8] Wikipedia. Non-functional requirements. http://en.wikipedia.org/wiki/Non-functional_requirement. Accessed: 2019-02-28.

[9] Georgios Yannakakis. Cellular automata for real-time generation of. https://www.researchgate.net/publication/228919622.

Appendix



# Source code

Here is some interesting code from the game:

## Chest.cs

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

public class Chest : MonoBehaviour
{
    [SerializeField] private Sprite chestOpenedSprite;
    [SerializeField] private SpriteRenderer spriteRenderer;
    [SerializeField] private SpriteRenderer itemSprite;
    public Sprite[] itemSprites;
    private bool isOpen = false;

    public void OnPlayerAttack()
    {
        if (!isOpen)
        {
            isOpen = true;
            spriteRenderer.sprite = chestOpenedSprite;

            int randomLoot = Random.Range(0, itemSprites.Length); // Obtener un n mero aleatorio para seleccionar el
            UpdatePlayerStats(randomLoot); // Actualizar las estadsticas del jugador seg n el bot n obtenido
            UpdateChestSprite(randomLoot); // Actualizar el sprite del cofre seg n el bot n obtenid
        }
    }

    private void UpdatePlayerStats(int lootIndex)
    {
        switch (lootIndex)
        {
            case 0: //ring_02
                GameController.MaxHealth += 2;
                GameController.HealPlayer(2);
                break;
            case 1: //feather_a
                GameController.MoveSpeed += 1f;
                GameController.Luck += 10;
                break;
            case 2: //sword_01
                GameController.AttackDamage += 1f;
                break;
            case 3: //clover_leaf
                GameController.Luck += 25;
                break;
            case 4: //spear_02
                GameController.AttackDamage += 0.5f;
                GameController.CriticDamage += 0.25f;
                break;
            case 5: //glasses
                GameController.CriticDamage += 0.25f;
                GameController.Luck += 10;
                break;
            case 6: //sword_02
```

```
                GameController.CriticDamage += 0.5f;
                break;
            case 7: //boots_02
                GameController.MoveSpeed += 1f;
                GameController.MaxHealth += 1;
                GameController.HealPlayer(1);
                break;
            case 8: //bow_02
                GameController.Luck += 10;
                GameController.AttackDamage += 0.5f;
                break;
            case 9: //bow_02
                GameController.MoveSpeed += 2f;
                break;
        }
        if (GameController.Luck > 100) GameController.Luck = 100;
    }

    private void UpdateChestSprite(int lootIndex)
    {
        // Verificar si el  dice  de botn est dentro del rango de sprites disponibles
        if (lootIndex >= 0 && lootIndex < itemSprites.Length)
        {
            itemSprite.sprite = itemSprites[lootIndex];
        }
        else
        {
            // Si el  dice  de botn est fuera de rango, mostrar un mensaje de advertencia
            Debug.LogWarning("Loot␣index␣out␣of␣range!");
        }
    }

    void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.tag == "Attack")
        {
            OnPlayerAttack();
        }
    }
}
```

## DungeonCrawler.cs

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

public class DungeonCrawler : MonoBehaviour
{
    public Vector2Int Position { get; set; }
    public DungeonCrawler(Vector2Int startPos)
    {
```

```
10            Position = startPos;
11        }
12
13        public Vector2Int Move(Dictionary<Direction, Vector2Int> directionMovementMap)
14        {
15            Direction toMove = (Direction)Random.Range(0, directionMovementMap.Count);
16            Position += directionMovementMap[toMove];
17            return Position;
18        }
19    }
```

### EnemyController.cs

```
1     using System.Collections;
2     using System.Collections.Generic;
3     using System.Transactions;
4     using Unity.VisualScripting;
5     using UnityEngine;
6     using UnityEngine.UIElements;
7
8     public enum EnemyState
9     {
10        Idle,
11        Wander,
12        Follow,
13        Attack,
14        Die
15    };
16
17    public class EnemyController : MonoBehaviour
18    {
19
20
21        public EnemyState currentState = EnemyState.Idle;
22
23        public float hp;
24        public float range;
25        public float speed;
26        public float attackRange;
27        public float attackFinishRange;
28        private bool isAttacking;
29        private bool isDamaged;
30        private bool chooseDir = false;
31        private bool dead = false;
32        private Vector3 randomDir;
33        public bool notInRoom = false;
34        private SpriteRenderer _spriteRenderer;
35        private Collider2D _collider;
36        private GameObject _player;
37        private Animator _animator;
38
39        void Start()
```

```
    {
        _player = GameController.player;
        _spriteRenderer = GetComponent<SpriteRenderer>();
        _animator = GetComponent<Animator>();
        _collider = GetComponent<Collider2D>();
}

    void Update()
    {
        switch(currentState)
        {
            case(EnemyState.Idle):
                Idle();
                break;
            case(EnemyState.Wander):
                Wander();
                break;
            case (EnemyState.Follow):
                Follow();
                break;
            case (EnemyState.Attack):
                Attack();
                break;
            case (EnemyState.Die):
                break;
        }

        if (!notInRoom)
        {
            if (IsPlayerInRange(range) && currentState != EnemyState.Die)
            {
                currentState = EnemyState.Follow;
            }
            else if (!IsPlayerInRange(range) && currentState != EnemyState.Die)
            {
                currentState = EnemyState.Wander;
            }

            if (_player.transform.position.x < transform.position.x)
            {
                _spriteRenderer.flipX = true;
            }
            else
            {
                _spriteRenderer.flipX = false;
            }

            if (Vector3.Distance(transform.position, _player.transform.position) <= attackRange && !isAttacking)
            {
                currentState = EnemyState.Attack;
            }

            if (gameObject.transform.position.y < _player.transform.position.y - 0.2f)
            {
```

```
94                  _spriteRenderer.sortingOrder = 2;
95              }
96              else
97              {
98                  _spriteRenderer.sortingOrder = 0;
99              }
100         }
101         else
102         {
103             currentState = EnemyState.Idle;
104         }
105     }
106
107     private bool IsPlayerInRange(float range)
108     {
109         return Vector3.Distance(transform.position, _player.transform.position) <= range;
110     }
111
112     private IEnumerator ChooseDirection()
113     {
114         chooseDir= true;
115         yield return new WaitForSeconds(Random.Range(3f, 5f));
116         randomDir = new Vector3(0, 0, Random.Range(0, 360));
117         chooseDir = false;
118     }
119
120     void Idle()
121     {
122
123     }
124
125     void Wander()
126     {
127         if(!chooseDir)
128         {
129             StartCoroutine(ChooseDirection());
130         }
131
132         transform.position += transform.right * speed * Time.deltaTime;
133         if (IsPlayerInRange(range))
134         {
135             currentState = EnemyState.Follow;
136         }
137     }
138
139     void Follow()
140     {
141         if (!isAttacking && !isDamaged)
142         {
143             transform.position = Vector2.MoveTowards(transform.position, _player.transform.position, speed * Time.del
144         }
145     }
146
147     void Attack()
```

```
    {
        if (!isAttacking && !isDamaged)
        {
            _animator.SetBool("Attack", true);
            isAttacking = true;
        }
    }

    void StopAttacking()
    {
        //switch(gameObject.name)
        //{
        //    case "Mushroom":
        //
        //        break;
        //}
        if (Vector3.Distance(transform.position, _player.transform.position) <= attackRange + attackFinishRange)
        {
            GameController.DamagePlayer(1);

        }

        _animator.SetBool("Attack", false);
        isAttacking = false;
    }


    public void Damaged()
    {
        hp -= GameController.Strike();
        if (hp <= 0)
        {
            Death();
        }
        _animator.SetBool("Damage", true);
        isDamaged = true;
    }

    public void EndDamage()
    {
        isDamaged = false;
        _animator.SetBool("Damage", false);
    }

    public void Death()
    {
        currentState = EnemyState.Die;
        _animator.SetTrigger("Death");
        _collider.enabled = false;
    }

    public void Destroy()
    {
        Destroy(gameObject);
```

```
202     }
203 
204     void OnTriggerEnter2D(Collider2D collision)
205     {
206         if (collision.tag == "Attack")
207         {
208             Damaged();
209         }
210     }
211 }
```

### GameController.cs

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.SceneManagement;
5  using UnityEngine.UI;
6  
7  public class GameController : MonoBehaviour
8  {
9      public static GameObject player;
10     public static int floorNumber = 0;
11     public Text statsText;
12 
13     public static GameController instance;
14     private static int health = 8;
15     private static int maxHealth = 8;
16     private static float moveSpeed = 6;
17     private static float attackDamage = 1;
18     private static float criticDamage = 1.5f;
19     private static int luck = 1;
20     public static bool fullStats = false;
21     private static Animator playerAnimator;
22     private AudioManager audioManager;
23 
24     public static int Health { get => health; set => health = value; }
25     public static int MaxHealth { get => maxHealth; set => maxHealth = value; }
26     public static float MoveSpeed { get => moveSpeed; set => moveSpeed = value; }
27     public static float AttackDamage { get => attackDamage; set => attackDamage = value; }
28     public static float CriticDamage { get => criticDamage; set => criticDamage = value; }
29     public static int Luck { get => luck; set => luck = value; }
30 
31     private void Awake()
32     {
33         Cursor.visible = false;
34         Cursor.lockState = CursorLockMode.Locked;
35         health = 8;
36         maxHealth = 8;
37         moveSpeed = 6;
38         attackDamage = 1;
39         criticDamage = 1.5f;
```

```
40        luck = 1;
41        player = GameObject.FindGameObjectWithTag("Player");
42        playerAnimator = player.GetComponent<Animator>();
43        if (instance == null)
44        {
45            instance = this;
46        }
47
48        audioManager = GameObject.FindGameObjectWithTag("Audio").GetComponent<AudioManager>();
49        audioManager.PlayMusic(audioManager.backgroundGame);
50    }
51
52    void Update()
53    {
54        if (fullStats)
55        {
56            statsText.text = "Spd:␣" + moveSpeed + "\n" +
57                             "Dmg:␣" + attackDamage + "\n" +
58                             "Crt:␣" + criticDamage + "\n" +
59                             "Lck:␣" + luck + "\n";
60        }
61        else
62        {
63            statsText.text = moveSpeed + "\n" +
64                             attackDamage + "\n" +
65                             criticDamage + "\n" +
66                             luck + "\n";
67        }
68
69    }
70
71    public static float Strike()
72    {
73        if (Random.Range(0, 101) <= luck)
74        {
75            return attackDamage * criticDamage;
76        }
77
78        else return attackDamage;
79    }
80
81    public static void DamagePlayer(int damage)
82    {
83        if (health >  0)
84        {
85            health -= damage;
86
87            if (health <= 0)
88            {
89                health = 0;
90                KillPlayer();
91            }
92        }
93    }
```

```

    public static void HealPlayer (int healAmount)
    {
        health = Mathf.Min(MaxHealth, health +  healAmount);
    }

    private static void KillPlayer()
    {
        playerAnimator.SetTrigger("Death");
    }

    public static void EndGame()
    {
        SaveHighestScore();
        SceneManager.LoadSceneAsync("MainMenu");
    }

    public static void SaveHighestScore()
    {
        if (PlayerPrefs.GetInt("HighestScore") < floorNumber)
        {
            PlayerPrefs.SetInt("HighestScore", floorNumber);
            PlayerPrefs.Save();
        }
    }
}
```

## MinimapCamera.cs

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

public class MinimapCamera : MonoBehaviour
{
    private Transform m_Camera;
    private Vector3 m_position;
    private float m_movSpeed = 2;

    private void Start()
    {
        m_Camera = GetComponent<Transform>();
    }
    private void Update()
    {
        if (CameraController.instance.currRoom != null)
        {
            m_position = new Vector3(CameraController.instance.currRoom.X * 0.32f, 5, CameraController.instance.currR
            m_Camera.position = Vector3.MoveTowards(m_Camera.position, m_position, Time.deltaTime * m_movSpeed);
        }
    }
```

```
24 }
```

## Room.cs

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using System.Linq;
4  using UnityEngine;
5
6  public class Room : MonoBehaviour
7  {
8
9      public int Width = 19;
10     public int Height = 11;
11     public int X;
12     public int Y;
13     private bool updatedDoors = false;
14     public bool roomVisited = false;
15
16     public Room(int x, int y)
17     {
18         X = x;
19         Y = y;
20     }
21
22     public GameObject minimapRoom;
23     public Door leftDoor;
24     public Door rightDoor;
25     public Door bottomDoor;
26     public Door topDoor;
27
28
29     public List<Door> doors = new List<Door>();
30
31     void Start()
32     {
33         if (RoomController.instance == null)
34         {
35             return;
36         }
37
38         Door[] ds = GetComponentsInChildren<Door>();
39         foreach (Door d in ds)
40         {
41             doors.Add(d);
42             switch (d.doorType)
43             {
44                 case Door.DoorType.right:
45                     rightDoor = d;
46                     break;
47                 case Door.DoorType.left:
48                     leftDoor = d;
```

```
                    break;
                case Door.DoorType.bottom:
                    bottomDoor = d;
                    break;
                case Door.DoorType.top:
                    topDoor = d;
                    break;
            }
        }
        minimapRoom = RoomController.instance.minimapRoomObject;
        RoomController.instance.RegisterRoom(this);
        minimapRoom.SetActive(false);
    }

    void FixedUpdate()
    {
        //EnemyController[] enemiesInRoom = GetComponentsInChildren<EnemyController>();
        List<GameObject> enemies = GetEnemiesInRoom(this, "Enemy");
        if (enemies.Count == 0)
        {
            RemoveConnectedDoors();
        }
        if(roomVisited)
        {
            minimapRoom.SetActive(true);
        }
        //else
        //{
        //    minimapRoom.SetActive(false);
        //}
    }

    public List<GameObject> GetEnemiesInRoom(Room parent, string tag)
    {
        List<GameObject> taggedChildren = new List<GameObject>();

        foreach (Transform child in parent.transform)
        {
            if (child.CompareTag(tag))
            {
                taggedChildren.Add(child.gameObject);
            }
        }

        return taggedChildren;
    }

    public void RemoveConnectedDoors()
    {
        foreach(Door door in doors)
        {
            switch (door.doorType)
            {
                case Door.DoorType.right:
```

```
103                     if (GetRight() != null)
104                         door.gameObject.SetActive(false);
105                     break;
106                 case Door.DoorType.left:
107                     if (GetLeft() != null)
108                         door.gameObject.SetActive(false);
109                     break;
110                 case Door.DoorType.bottom:
111                     if (GetBottom() != null)
112                         door.gameObject.SetActive(false);
113                     break;
114                 case Door.DoorType.top:
115                     if (GetTop() != null)
116                         door.gameObject.SetActive(false);
117                     break;
118             }
119         }
120     }
121 
122     public void CloseDoors()
123     {
124         foreach (Door door in doors)
125         {
126             door.gameObject.SetActive(true);
127         }
128     }
129 
130     public Room GetRight()
131     {
132         if (RoomController.instance.DoesRoomExist(X + 1, Y))
133         {
134             return RoomController.instance.FindRoom(X + 1, Y);
135         }
136         return null;
137     }
138     public Room GetLeft()
139     {
140         if (RoomController.instance.DoesRoomExist(X - 1, Y))
141         {
142             return RoomController.instance.FindRoom(X - 1, Y);
143         }
144         return null;
145     }
146     public Room GetTop()
147     {
148         if (RoomController.instance.DoesRoomExist(X, Y + 1))
149         {
150             return RoomController.instance.FindRoom(X, Y + 1);
151         }
152         return null;
153     }
154     public Room GetBottom()
155     {
156         if (RoomController.instance.DoesRoomExist(X, Y - 1))
```

```
157         {
158             return RoomController.instance.FindRoom(X, Y - 1);
159         }
160         return null;
161     }
162 
163     private void OnDrawGizmos()
164     {
165         Gizmos.color = Color.red;
166         Gizmos.DrawWireCube(transform.position, new Vector3(Width, Height, 0));
167     }
168 
169     public Vector3 GetRoomCentre()
170     {
171         return new Vector3(X * Width, Y * Height);
172     }
173 
174     private void OnTriggerEnter2D(Collider2D other)
175     {
176         if(other.tag == "Player")
177         {
178             RoomController.instance.OnPlayerEnterRoom(this);
179         }
180     }
181 }
```

## SlimeBoss.cs

```
1  using System;
2  using System.Collections;
3  using UnityEngine;
4  using UnityEngine.InputSystem.Processors;
5  using UnityEngine.UIElements;
6  
7  public class SlimeBoss : MonoBehaviour
8  {
9      public float hp = 20f;
10     public float jumpForce = 30f;
11     public float jumpCooldown = 5f;
12     public float jumpHeight = 2f;
13 
14     private Room room;
15     private TransitionController transitionController;
16     private Rigidbody2D rb;
17     private Transform transform;
18     private Vector3 playerDirection;
19     private Animator animator;
20     private bool isJumping = false;
21     private bool canJump = true;
22     private bool isDamaged = false;
23     private bool bossActive = false;
24     private static Transform playerTransform;
```

```
25      private SpriteRenderer spriteRenderer;
26
27      void Start()
28      {
29          playerTransform = GameController.player.transform;
30          transform = GetComponentInParent<Transform>();
31          animator = GetComponentInParent<Animator>();
32          room = GetComponentInParent<Room>();
33          transitionController = room.GetComponentInChildren<TransitionController>();
34          rb = GetComponent<Rigidbody2D>();
35          spriteRenderer = GetComponent<SpriteRenderer>();
36      }
37
38      void Update()
39      {
40          if (bossActive)
41          {
42              if (!isJumping && canJump && bossActive)
43              {
44                  StartJump();
45              }
46
47              if (playerTransform.position.x < transform.position.x)
48              {
49                  spriteRenderer.flipX = true;
50              }
51              else
52              {
53                  spriteRenderer.flipX = false;
54              }
55
56              //if (isJumping)
57              //{
58              //
59              //    transform.position = Vector2.MoveTowards(transform.position, playerDirection, jumpForce * Time.deltaTime);
60              //
61              //}
62
63              if (gameObject.transform.position.y < playerTransform.position.y - 0.2f)
64              {
65                  spriteRenderer.sortingOrder = 2;
66              }
67              else
68              {
69                  spriteRenderer.sortingOrder = 0;
70              }
71          }
72          else if (room == CameraController.instance.currRoom)
73          {
74              StartCoroutine(BossActive());
75          }
76      }
77
78      IEnumerator BossActive()
```

```
79      {
80          yield return new WaitForSeconds(0.4f);
81          room.CloseDoors();
82          yield return new WaitForSeconds(0.4f);
83          bossActive = true;
84      }
85
86      void StartJump()
87      {
88          playerDirection = playerTransform.position;
89          canJump = false;
90          animator.SetBool("Jump", true);
91      }
92
93      void Jump()
94      {
95          isJumping = true;
96          Vector2 direction = (playerDirection - transform.position).normalized;
97          rb.velocity = direction * jumpForce;
98      }
99
100     void FinishJump()
101     {
102         isJumping = false;
103         rb.velocity = Vector2.zero;
104         animator.SetBool("Jump", false);
105         Invoke("EnableJump", jumpCooldown);
106     }
107
108     void EnableJump()
109     {
110         canJump = true;
111     }
112
113     void Damaged()
114     {
115         if (!isJumping && !isDamaged)
116         {
117             animator.SetBool("Damaged", true);
118             isDamaged = true;
119             hp -= GameController.Strike();
120             if (hp <= 0)
121             {
122                 Death();
123             }
124         }
125     }
126
127     void FinishDamage()
128     {
129         animator.SetBool("Damaged", false);
130         isDamaged = false;
131     }
132
```

```
133     void Death()
134     {
135         animator.SetTrigger("Death");
136     }
137 
138     void Destroy()
139     {
140         Destroy(gameObject);
141         transitionController.Open();
142     }
143 
144     void OnTriggerEnter2D(Collider2D collision)
145     {
146         if (collision.tag == "Attack")
147         {
148             Damaged();
149         }
150     }
151 
152     private void OnCollisionEnter2D(Collision2D collision)
153     {
154         if (collision.gameObject.tag == "Player")
155         {
156             GameController.DamagePlayer(2);
157         }
158     }
159 }
```

## TransitionController.cs

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using Unity.VisualScripting;
4  using UnityEngine;
5  
6  public class TransitionController : MonoBehaviour
7  {
8      [SerializeField] private SpriteRenderer spriteRenderer;
9      [SerializeField] private Sprite openedTrapdoor;
10     private bool isOpened = false;
11 
12     public void Open()
13     {
14         isOpened = true;
15         spriteRenderer.sprite = openedTrapdoor;
16     }
17 
18     void OnTriggerEnter2D(Collider2D collision)
19     {
20         if (collision.tag == "Player" && isOpened)
21         {
22             GameController.floorNumber += 1;
```

```
23              RoomController.instance.GetComponent<DungeonGenerator>().ClearDungeon();
24
25              ResetPlayerPosition();
26          }
27      }
28
29
30      public void ResetPlayerPosition()
31      {
32          GameController.player.transform.position = Vector3.zero;
33          CameraController.instance.transform.position = new Vector3(0,0,-10);
34
35          if (RoomController.instance != null)
36          {
37              Room currentRoom = RoomController.instance.GetComponent<RoomController>().startRoom;
38              RoomController.instance.GetComponent<RoomController>().OnPlayerEnterRoom(currentRoom);
39          }
40      }
41  }
```