POLISH-JAPANESE ACADEMY
OF INFORMATION TECHNOLOGY

Design and Development of the "Card your Time" Hybrid Rougelike Card Game in Unity

Zakharii Zubyk S24984
Maksymilian Janas S25135
Juliusz Kupiński S23751
Jan Kajszczak S22714

Polish-Japanese Academy of Information Technology
Faculty: Computer Science
Department: eXtended Reality and Immersive Systems
Specialized area: eXtended Reality, Games and Immersive Systems
Thesis prepared under the supervision of:
Kinga Skorupska, PhD
Reviewer:
Barbara Karpowicz, MScEng

Warsaw, March 23, 2025

POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Projekt i przebieg pracy nad grą: “Card your time”. Hybryda Roguelike i gry karcianej w Unity.

Zakharii Zubyk S24984
Maksymilian Janas S25135
Juliusz Kupiński S23751
Jan Kajszczak S22714

Polsko-Japońska Akademia Technik Komputerowych
Wydział: Informatyka
Katedra: eXtended Reality i Systemów Immersyjnych
Zakres specjalistyczny: eXtended Reality, Gry i Systemy Immersyjne
Praca dyplomowa przygotowana pod opieką:
dr Kinga Skorupska
Recenzent:
mgr inż. Barbara Karpowicz

Warszawa, 23 marca 2025

Abstract. This thesis outlines the development process of a computer game combining elements of a **turn-based card game** and a **roguelike**. In the game, the player travels through **randomly generated rooms**, meeting **various characters and opponents**. The key mechanics are the **time system per turn** and **combining equipment with crafting a deck of cards**. Throughout the game, players explore various locations, fight opponents, develop their characters and collect cards and items. The dungeon creation is based on templates in which each room is randomly selected from a chosen bundle. Throughout the project development, many tools were used. One of them is the Unity Game Engine, being the most crucial part of game development. Other tools include IDEs such as Visual Studio or Rider, and graphical programs, in which the most prominent was Aseprite. The creation process was based on Agile/Scrum methodology. There was also a risk analysis and a plan to manage any issues such as team abandonment or graphics problems.

Keywords: Turn-based · Cards game · Roguelike.

Streszczenie Projekt dotyczy gry komputerowej, łączącej elementy **karcianej gry turowej** i **roguelike**. W grze, gracz przemierza **losowo generowane pokoje**, spotykając **różne postaci i przeciwników**. Kluczowe mechaniki to **system czasu w turze** oraz **połączenie ekwipunku z tworzeniem talii kart**. W trakcie rozgrywki gracze eksplorują różne lokalizacje, walczą z przeciwnikami, rozwijają swoje postacie i zbierają karty oraz przedmioty. Tworzenie dungeonu bazuje na szablonach w których każdy pokój jest losowany z wybranej grupy. W trakcie projektu używane było wiele narzędzi. Jednym z nich jest silnik Unity, będący kluczową częścio tworzenia gry. Do innych narzędzi zaliczają się IDE takie jak Visual Studio lub Rider, oraz programy graficzne, na przykład Aseprite.Proces tworzenia gry bazował na metodyce Agile/Scrum. Dokument zawiera również analizę ryzyk oraz plany jak radzić sobie z problemami, na przykład opuszczenie zespołu czy problemy z grafiką.

Keywords: Turowość · Gra karciana · Roguelike.

Table of Contents

1 Introduction

In the realm of gaming, the fusion of various genres often sparks innovation and excitement. Our endeavor embarks on this path, seeking to blend the strategic depth of card games with the immersive challenges of turn-based gameplay and the unpredictable thrill of roguelike adventures.

1.1 Goals of the Work

Our aim for this project was to develop an immersive and demanding video game incorporating elements from card games, turn-based strategies, and roguelikes.

1.2 Motivation

The inspiration for this project stems from a deep appreciation for the intricacies of game design and the desire to create an experience that captivates players across diverse gaming preferences. We aim to craft a game that not only entertains but also challenges players, pushing them to strategize, adapt, and overcome in a dynamic, ever-changing environment.

Drawing from the rich tapestry of card game mechanics, the structured decision-making of turn-based strategy, and the procedural generation of roguelikes, our motivation lies in crafting a gaming experience that offers depth, replayability, and a sense of accomplishment with each playthrough.

With the utilization of modern technologies and innovative approaches, we aim to realize a vision that not only meets but exceeds the expectations of players, delivering an unforgettable journey into the depths of the unknown.

2 Game Design

Unique parts of our game consists of a story and mechanics such as a time system, and system of deck creation joined with a more standard equipment system. Another unique element consists of randomly generated dungeons for a unique experience with each playthrough.

2.1 General Idea, Genre and Core Mechanics

2.1.1 Turn-based timeline

Every card apart from certain exceptions has its cost. Cost varies between cards.The higher the cost of the cards you use, the longer you will have to wait for your next turn. The cost of the cards adds up to the clock next to the character. Every character in the game has their own independent clock. When the turn of a character is finished the "timeline" starts. Every second one "second" from the clock is subtracted, when it reaches zero the turn of this character starts. For example: if a player uses the card of a cost 10 and enemy1 uses the card of a cost 15, then the player's next turn will occur first.

2.1.2 Grid system

Each character is located in a 2D matrix called a grid. The grid resembles a chessboard where every character has its own field where it stands. The whole grid does not have to be square sized like a chessboard. Depending on its action, all characters can change position on a grid and interact with fields occupied by other characters.

2.1.3 Card system

Combination deck-building mechanics with traditional RPG elements, allowing players to create diverse strategies by customizing their decks and equipment load outs. Adaptability, planning, and understanding of card interactions are key to success in battles.

2.1.4 Equipment

Equipping different gear alters the cards available to a character. For example, equipping a new weapon might add or change the attack cards in your deck. As characters level up or acquire better equipment, they gain access to more powerful cards, enhancing their

capabilities in battles. Equipment can provide good cards, bad cards and cursed cards (black one).

2.1.5 General idea

The genres of the game consist of marrying elements of Roguelike, such as random generated dungeons and equipment creation, with a deck-builder. The game loop is based on turns, enemy and player turns are given priority based on the value of their clocks.

Each time you start the game, a dungeon is procedurally generated, featuring rooms and corridors. Players begin with a set of equipment that provides cards (see Fig. 8) to use during battles. The objective is to navigate through the dungeon, improve your gear, and survive while progressing toward a portal room.

The portal room serves as a checkpoint, advancing the player deeper into the game. However, most rooms encountered will typically involve combat (some lore dialog room). Upon entering a new, unexplored room, the player engages in a fight (see Fig. 2).

In scenarios where two or more clocks display the same value, a predefined priority system determines the outcome (see Fig. 1). Additionally, clock values are restricted to a range between 0 and 99, ensuring consistency and preventing out-of-bound values.

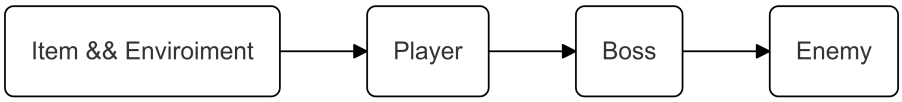


Fig. 1: Graph with priority who go first if more than 1 clocks have 0 in our Clock System
Source: own elaboration

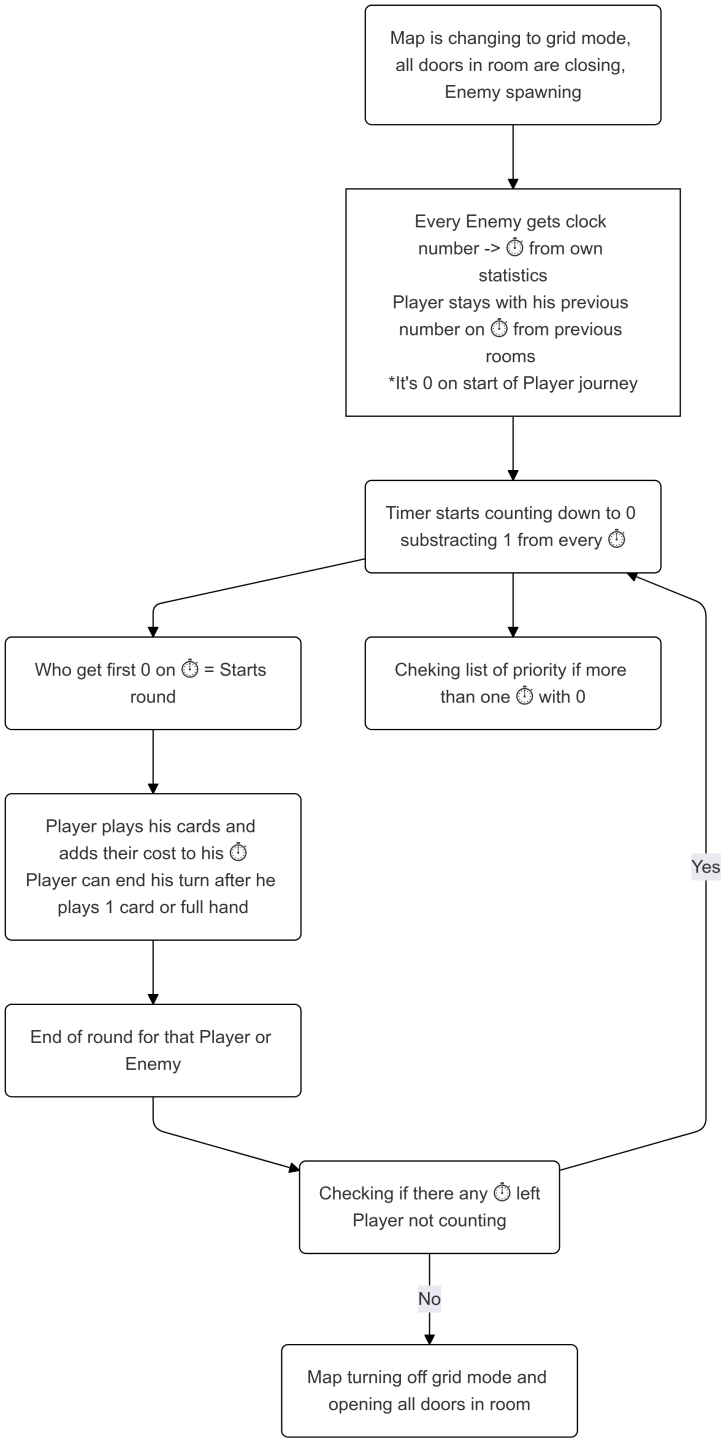
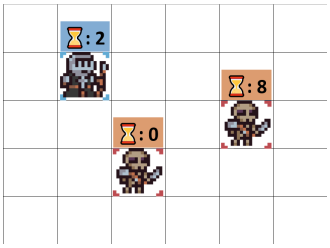
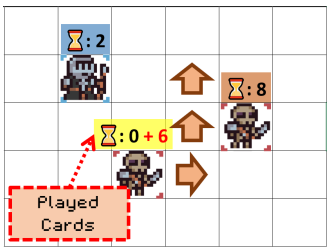


Fig. 2: Graph with logic how work our Clock System
Source: own elaboration

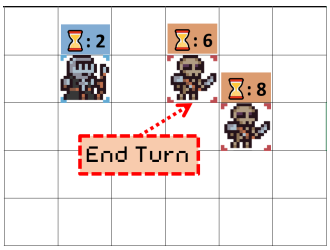


1) Player entered room with enemy's.

2) Who have 0 on ◷(Clock) start's round (right now it's Enemy).



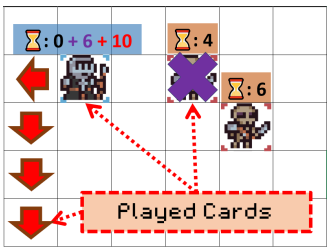
3) Enemy play's cards which increase his. ◷



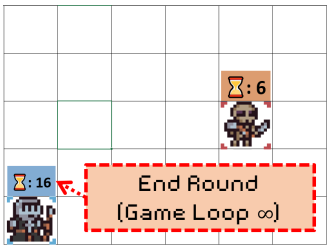
4) Enemy end his round.

5) Manager start's counting down ◷'s.

6) Who gets 0 on ◷start's round (right now it's Player).



7) Player play's cards and increase his ◷.



8) Player end his round. And it will repeat until all enemies dies or player will die.

Fig. 3: Explanation of Timers
Source: own elaboration

2.2 Research and Inspiration

The design process for a game combining various elements of cards, turn-based mechanics, and roguelike gameplay draws inspiration from various sources, both classic and innovative.

- **Card-based mechanics** offer a dynamic and strategic layer, highlighting deck-building, resource management, and tactical decision-making. These systems are inspired by traditional card games and modern digital adaptations, focusing on creating significant choices with each draw or play. (Wildfost (see page 15); Card Hunter (see page 17))
- **Turn-based gameplay** brings unhurried decisions, allowing players to plan their actions carefully. This format enhances strategy and provides a fair challenge, where success relies on forethought rather than reflexes. It takes best from classic turn-based RPGs and tactical games, encouraging a sense of control and mastery.(Into The Breach (see page 19); Crown Trick (see page 23); Metal Gear AC!D (see page 21))
- **The roguelike genre** adds replayability and unpredictability through procedurally generated content and permadeath mechanics. These elements challenge players to adapt to ever-changing scenarios, making each run feel unique and rewarding persistence. The inspiration systems from the early roguelike pioneers were reimagined to fit the evolving demands of modern gaming. (Enter The Gungeon (see page 24); Binding of Isaac (see page 24))

This combination creates a compelling blend of strategy, adaptability, and creativity, encouraging players to experiment with their approaches while embracing the thrill of uncertainty.

2.2.1 Card-Based Mechanics

Wildfrost

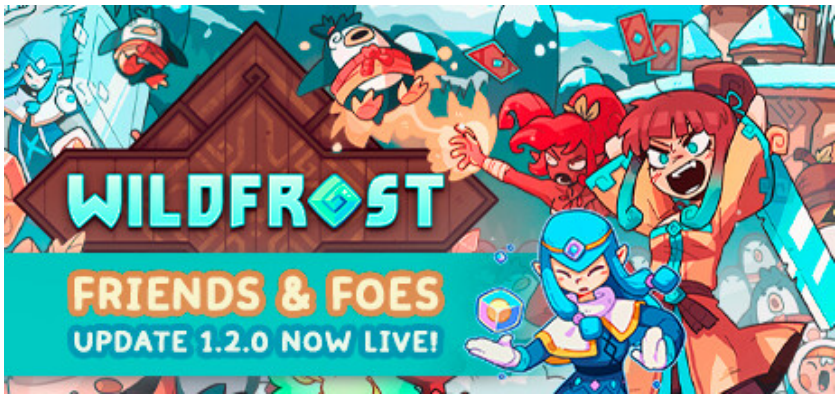


Fig. 4: Logo of "Wildfrost"
Source[1]

The card format (see Fig. 4) in Wildfrost (see Fig. 5) blends visual style, intuitive icons, and strategic elements to make each card readable and engaging. Key components include visually distinct stats for attack, health, and countdowns, color and style selection varying between different types of cards, straightforward rarity indicators, direct and simple text for each card's description, and art style that perfectly aligns with the game's theme.

[1] https://store.steampowered.com/app/1811990/Wildfrost/
[2] https://wildfrostwiki.com/Chungoon

Fig. 5: Example of in-game card from the “Wildfrost” Source[2]

Card Hunter

Fig. 6: Logo of the game "Card Hunter"
Source[1]

Card Hunter (see Fig. 6) is the second main inspiration for the game due to the tight connection between the character's equipment and the deck-building system. Each piece of gear - like weapons, armor, or trinkets - comes with its own set of cards (see Fig. 8). Instead of collecting cards directly, the player adjusts his deck by selecting the gear that brings his desired cards along with it. The card design (see Fig. 7) is also inspiring because it balances clarity with a vintage-inspired aesthetic, connecting both the mechanics and the visual charm of tabletop games to create an engaging and immersive experience for players.

[1] https://en.wikipedia.org/wiki/Card_Hunter
[2] http://wiki.cardhuntria.com/wiki/Cards
[3] https://www.armchairdragoons.com/articles/reviews/classic-reviews-card-hunter-originally-by-blue-manchu-games/

Fig. 7: Example of card from "Card Hunter"
Source[2]

Fig. 8: Example of deck building via items in "Card Hunters"
Source[3]

2.2.2 Turn-Based Gameplay

Into The Breach

Fig. 9: Logo of "Into The Breach"
Source[1]

Into the Breach (see Fig. 9) keeps the gameplay engaging by combining repeating map layouts and objectives with highly dynamic, turn-by-turn tactics and procedurally generated challenges. Each mission requires you to reassess your strategy, leverage the terrain, and maximize your squad's unique abilities to survive. This blend of predictability in objectives with variability in tactics and enemy actions ensures that no two games feel exactly alike, making each run through the game feel like a fresh strategic puzzle (see Fig. 10).

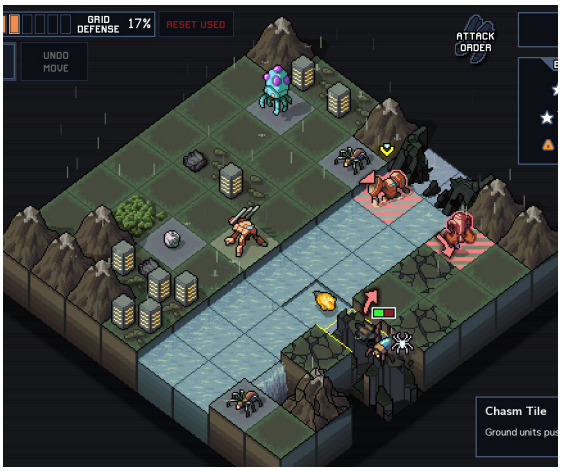


Fig. 10: Example of gameplay with repetitive objects in "Into The Breach" Source[2]

[1] https://store.steampowered.com/app/590380/Into_the_Breach/
[2] https://store.steampowered.com/app/590380/Into_the_Breach

Metal Gear AC!D

Fig. 11: Logo of "Metal Gear Ac!d" Source[1]

Metal Gear Acid was the main inspiration for the project influencing its core mechanics and the view of the gameplay. The biggest inspiration was the system of turns structured around a "clock" (see Fig. 13), where every action taken by a character - be it movement, attacking, or using a card - costs a set number of points. All accumulated points are added to the character's "clock", determining when the character can act again. This creates a layer of strategy, as the player needs to balance aggressive moves with defensive positioning. He has to ensure, he can react to threats without becoming too vulnerable during the opponent's turn. In Metal Gear Acid environment plays a significant tactical during combat as a result each map is also a strategic puzzle that connects stealth mechanics with the turn-based card system (see Fig. 11). The card design serves a dual purpose (see Fig. 12). They have to be visually distinct to support fast decision-making but also to engage players with intricate, meaningful designs that capture the military, espionage-heavy spirit of the Metal Gear franchise.

Fig. 12: Example of in-game card from the "Metal Gear Ac!d" Source[2]

Fig. 13: Example of gameplay with timer in "Metal Gear Ac!d" Source[3]

Crown Trick

Fig. 14: Logo of "Crown Trick"
Source[1]

Crown Trick is a grid-based, turn-based RPG (see Fig. 15). The unique interplay of grid-based movement, tactical turn-based choices, and environmental interaction makes it a deeply strategic game. Each move feels consequential, and the turn-based structure means that every step you take can bring either reward or risk, making each encounter in the dungeon an intense and satisfying puzzle.

Fig. 15: Example of gameplay grid turn-based in "Crown Trick"
Source[1]

[1] https://en.wikipedia.org/wiki/Metal_Gear_Acid
[2] https://metalgear.fandom.com/wiki/Metal_Gear_Acid_cards
[3] https://pixelhunted.wordpress.com/tag/psp/

2.2.3 The Roguelike Genre

Enter the Gungeon and Binding of Isaac

Fig. 16: Logo of "Enter the Gungeon"
Source[1]

Fig. 17: Logo of "Binding of Isaac"
Source[2]

Both Enter the Gungeon and Binding of Isaac use room-based dungeon generation, where each floor is divided into rooms that are procedurally arranged (see Fig. 18). Rather than generating every tile randomly, these games create each room with a specific purpose (e.g., treasure rooms, boss rooms, or shops) and then connect them in varying configurations. In The Binding of Isaac (see Fig. 17), each room template has a predefined configuration of obstacles, enemies, events (see Fig. 19), and item placements, which are randomized during dungeon generation. Enemies and items are slightly changed depending on the game progression and difficulty level, to keep the game exciting and unpredictable.

[1] https://store.steampowered.com/app/1000010/Crown_Trick/
[1] https://store.steampowered.com/app/311690/Enter_the_Gungeon/
[2] https://store.steampowered.com/app/113200/The_Binding_of_Isaac/
[3] https://tech-en.netlify.app/articles/en519658/
[4] https://www.youtube.com/watch?v=XHUS0KRc_EY

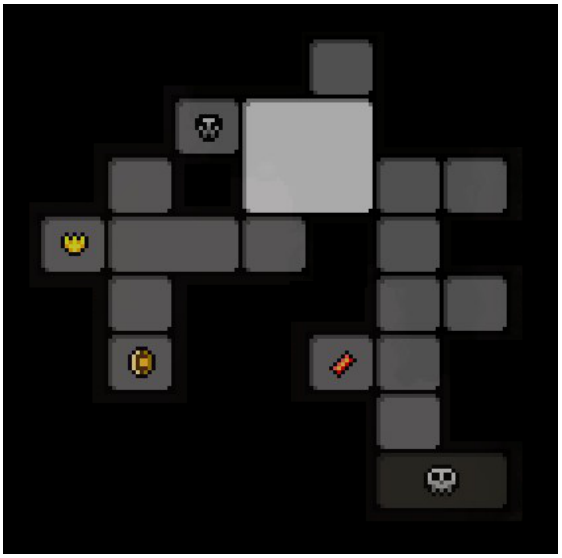


Fig. 18: Example of generation level in "Binding of Isaac"
Source[3]

Fig. 19: Example of an in-game event of the "Enter the Gungeon"
Source[4]

Neophyte

Fig. 20: Logo of "Neophyte"
Source[1]

The GUI (see Fig. 21) in Neophyte (see Fig. 20) is streamlined and purposeful, offering a well-balanced approach that prioritizes readability and accessibility. By focusing on contrast (semi-opacity), simplicity (simple icons), and strategic placement: items at the bottom left corner, stats at the bottom right corner, character and effects at the bottom middle, etc. The interface supports the gameplay without detracting from the experience, allowing players to fully engage with the game world and react quickly to in-game events.

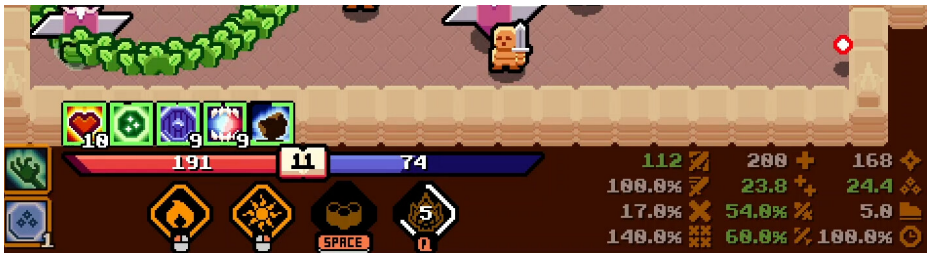


Fig. 21: Example of GUI in mid-gameplay of the "Neophyte"
Source[1]

[1] https://store.steampowered.com/app/1409530/Neophyte/

Tiny Rogues

Fig. 22: Logo of "Tiny Rogues"
Source[1]

Risk and Reward system (see Fig. 23)) in Tiny Rogues (see Fig. 22) is designed to be dynamic, forcing players to balance immediate gains against future risks while managing unpredictable elements and adapting to evolving challenges. This structure gives each run a unique feel and creates a strategic depth that keeps players engaged as they decide how far they're willing to push their luck for the promise of powerful rewards.

Fig. 23: Example of in-game buff for Player in "Tiny Rogues"
Source[2]

Moonlighter

Fig. 24: Logo of "Moonlighter"
Source[1]

In Moonlighter Each weapon type offers a specific rhythm and strategy (see Fig. 25) that impacts how players handle dungeons and interact with different types of enemies. For instance, the sword and shield's versatility might encourage more exploration and experimentation, while the gloves demand precision and quick reactions. Similarly, the sword and the spear reward players who can read enemy patterns and manage distance effectively.

[1] https://store.steampowered.com/app/2088570/Tiny_Rogues/
[2] https://steamcommunity.com/app/2088570/allnews/?l=danish

Fig. 25: Example of different playstyles in "Moonlighter"
Source[2]

In Moonlighter Each weapon type offers a specific rhythm and strategy (see Fig. 25) that impacts how players handle dungeons and interact with different types of enemies. For instance, the sword and shield's versatility might encourage more exploration and experimentation, while the gloves demand precision and quick reactions. Similarly, the sword and the spear reward players who can read enemy patterns and manage distance effectively.

2.3 Mechanics

The list of game mechanics and possibilities in the game consists of:

- Exploring randomly generated locations inside the dungeon.
- Fighting against opponents in a turn-based system with a timer.
- Collecting items and equipment found in the game world.
- Items and equipment used by the player determine what cards the player can draw and use during combat.

[1] https://store.steampowered.com/app/606150/Moonlighter/
[2] https://www.youtube.com/watch?v=uGz7ltNCAz0

3 Related Works

3.1 Player Experience and Game Design

Game design is one of the most important parts of the game development process. During it, developers look for ways to create an enjoyable experience for all players who might be interested in the game. This project is no different.

The first paper considered for the research on game design is called "MDA: A Formal Approach to Game Design and Game Research"[8]. It outlines a formal and iterative approach to game design that seeks to keep all players invested in the game for as long as it can. The paper divides the design process into three parts. It also shows the differences in how the game is seen and experienced between players and developers. The paper states that an approach to game design should not be driven only by a desire for more interesting features, but rather by the experience the developer wants to give to the players. Lastly, the paper outlines its "three layers of abstraction" which helps to understand games more as a dynamic system.

Another source is the "Understanding the Psychology of Game Design"[4]. The book uses "The Wither 3" as an example based on which it outlines how a developer can understand the psychology of players, and shows how those findings my help in the process of game design. This work is instrumental in creating game mechanics that are not only engaging but also psychologically rewarding for players.

The article "Game Feel: A Beginner's Guide"[5] describes, what is game feel, and how to use it in game development. The source is an essential way to learn about principles of game design from the point of simple button inputs, ending on narrative, and the overall look of the game. The article focuses on terms like responsiveness, intuitiveness, and viscerality. The paper also outlines how the concept of game feel affects the player's experience of the game.

The paper "Leveling up fun: Learning progress, achievement, and expectations influence enjoyment in video games"[6] answers a question of what factors influence how much fun people have while completing enjoyable tasks. The study is focused on large video games. It has concluded that most people have the most fun in games of average difficulty, though additional factors have also shown themselves. The paper states that the player's enjoyment is strictly tied

to the difficulty of the game, and also ties the enjoyment with an ability to learn better. It also shows that the enjoyment is tied to the ability of the player to predict how hard the game will be based on the difficulty settings. The paper also talks about several other factors influencing player enjoyment of video games.
In the search for a better understanding of game design, one of the best sources turned out to be the "Unity" engine guides. One of those guides called "User interface design and implementation in Unity"[12] outlines ways in which a reliable UI system can be created using two main tools available for the developers in the "Unity" game engine, those being the UI Toolkit and Unity UI. The guide also walks its users through the process of creating their fonts, preparing assets, and the UI design and development phases.

3.2 Procedural Generation

Throughout the research process on the topics related to random generation in games, most of the related works focused on "roguelikes" due to the randomness of each new playthrough associated with them. The "Analysis and development of a game of roguelike genre"[14] drew attention to itself, by going deeply into the roguelike genre, focusing on the procedural generation of the game content (e.g., generating dungeons in dungeon crawler type games). It goes through many types of procedural generation, from generation types based mostly on algorithms to those that generate content using premade templates. This analysis is crucial for understanding how to implement and optimize procedural generation techniques.
"Procedural Content Generation in Games" [13] is one of the most detailed analyses of procedural generation in games. The text explores the history of the development of procedural content generation and its application in various games like StarCraft and Infinite Mario Bros. It examines various techniques for procedurally generating game content. Using many examples, each technique is described in detail.
The last work to mention is "The ANGELINA Videogame Design System, Part 1." [7] This article presents ANGELINA, an autonomous coevolutionary system. Unlike the previous works that discussed generating separate parts of games, ANGELINA creates whole, com-

plete, and playable games. The system is a great step in the area of procedural content generation. It adapts multiple creative processes like level design, ruleset creation, and aesthetic theming to complement each other dynamically.

3.3 Game Development Techniques

In the beginning and the middle stages of project development documents about the developing techniques started to catch our attention. The main purpose of these techniques is to optimize the process of the game creation. The first one to mention is "How Are Agile Methods and Practices Deployed in Video Game Development? A Survey into Finnish Game Studios" [10]. It's a study investigating the use of agile methods and practices in game development inside Finnish game studios. The investigation reveals that the most used agile frameworks are Scrum, XP (eXtreme Programming), and Kanban because they address the complex and iterative nature of game development. The survey conducted in the game studios shows that the most applied framework is Scrum. Its practices such as sprints, daily scrums, and cross-functional teams improve communication, the quality of the game, and help with coming up with and implementing "funny" features. However, it doesn't eliminate some challenges like overworking and feature creep.
The second work is "Level up your programming with game programming patterns" [2]. An e-book released by Unity created as a guide for game developers. It focuses on applying SOLID (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) principles while developing a code side of video game projects. Additionally, it describes advanced design patterns such as Factory pattern, Object pool, Singleton pattern, Command pattern, State pattern, and Observer pattern by providing examples that represent popular problems in game development. For explanation, it uses specific tools, which are a part of the Unity game engine, which makes it a valuable source for developers working with the engine.
The third and last work to mention is another e-book released by Unity titled "User interface design and implementation in Unity" [12]. This topic is completely different from the previously men-

tioned e-book as it focuses on the connection between artistic design and technical implementation in User Interface (UI) development in Unity. UI is a crucial aspect of all games, especially if they need to provide key information to the player without using any text. It emphasizes the importance of visual aesthetics and the functionality of UI design in games. It presents key practices like wireframing, prototyping, and asset optimization, along with the tools that support these practices, such as Sprite Editor, PSD Importer, and UI Builder.

3.4 Card and Strategy Games

In the development and design of card games within the realm of game development, several works have been influential. These works provide a foundation and context for our own research and development efforts. Below we highlight and discuss some of the most relevant studies and articles, and their contributions to the field.
The first article we would like to mention is "A Guide to Designing and Self-Publishing Card Games" [1]. In this work, the author presents us with a deep and detailed description of the elements behind the creation of successful card sets from both a technical and a marketing perspective. The article is a rich collection of tips and issues related to designing card games - the lessons from it have proven valuable in adapting the overall vision of our video game.
Organization and documentation are essential factors when designing cards for a card video game, especially for titles with a wide list of different cards. "Building a Home: How to Construct a Card Game" [9] shows an outline of a card game development process, from creating a Card File, through playtesting, to balancing and art design. The author of the article guides the reader through every step of the process, providing detailed information and insights. It is a source of practical knowledge for game designers who want a stable and efficient workflow to help them create engaging card games.
"Methods for Card Drafting" [3] is the title of a blog posted on the BoardGameGeek webpage, where board game and card game enthusiasts can discuss topics related to different aspects of board games. In this post, the author identifies and describes different methods for card drafting, such as the "Blind Draw", "Auction", or "Bingo"

method. Other discussion participants share their knowledge and findings related to the topic, thus creating a point of research for future game developers.
Finally, “Adaptive Spatial Reasoning for Turn-Based Strategy Games” [11] is a research paper that discusses the author’s proposal for a game AI architecture, called ADAPTA. The architecture described in this paper was designed for efficient design of AI systems for Turn-Based Strategy games - it is further proved through various experiments, showing its potential in outperforming static opponents and capabilities in generating AIs. This scientific study and its insights are suitable for developing AI adversaries in our game.

4 Tools and methods

Descriptions of tools used during completion of the project, and of ways in which they were used with a justification for the choice of each tool.

4.1 Unity

Video game engine previously known to every team member based on previously done projects.

4.1.1 Cinema machine A Unity package for advanced camera control, enabling dynamic, cinematic shots with ease, such as tracking, framing, and smooth camera transitions. A tool simplifying the implementation of in-game cameras on a very high level. We use it for a better user experience with Player and Enemy cameras. The cinemachine is responding for a smooth transition from one object to another and a smooth camera follows.[1]

4.1.2 Toast Notification Message The Toast Notification Message is a Unity plugin that provides a simple and effective way to display toast-style messages in your game. With this plugin, developers can easily show short and informative messages anywhere on the screen. Messages can be customized with text, icons, and more! Text from official page.[2] We used it for implementation of EventEnemy and Tutorial (see 5.19). We had to hard customize this plugin so it fit our game.

4.1.3 Tactics Toolkit: Pathfinding A Unity package that supports the creation of tactical and strategy games, offering ready-made systems for grid-based movement and pathfinding. We took an idea to create overlays for each tile of the grid based map and adjusted it to make it compatible with our map generation system. It made our game more interactive for the player and more visually appealing. We adjusted a range finding system and algorithm so it

[1] https://unity.com/unity/features/editor/art-and-design/cinemachine

[2] https://assetstore.unity.com/packages/tools/gui/toast-notification-message-278006?srsltid=AfmBOopd7fKEw1OG6fQTXVSEOpy0wlHLDOQ-YnaQW1FeSjMzkPGyrhSk

works with the cards and enemies and adjusted it so it counts out the range of the card abilities. We also adjusted a path finding alghorythm so it counts out the path for the player's movement depending if he's in combat or not.[3]

4.1.4 Unity-card-play A Github repository with the MIT licnece created by Alexandru Apostu offering simple solution for implenting the player's card hand. Base idea of the project was massively altered to make it compatible with our project. It made the interaction with the card hand pleasant and smooth and was a massive improvemt to the first prototype.[4]

4.1.5 Addressables A Unity package giving its users an easy way to access assets during gameplay. The plugin also supports loading assets from a cloud environment. At first the "AssetsDatabase" class was used for the same purpose, but it has turned out that the class can only be accessed in editor, not in the game itself.[5]

4.1.6 Input system A Unity package now now being part of the engine, but in this game had to be implemented by hand. The package makes implementation of player controls much easier and centralized, giving the developers much more control.[6]

4.2 IDE

4.2.1 Visual Studio A widely used IDE by Microsoft for coding, debugging, and compiling applications, especially popular in .NET and C# development. Some of us are using Visual Studio (see Fig. 26) as main IDE for programming C# scripts for Unity.[7]

[3] https://assetstore.unity.com/packages/templates/tutorials/tactics-toolkit-pathfinding-237954
[4] https://github.com/lelexy100/unity-card-play
[5] https://docs.unity3d.com/Packages/com.unity.addressables@2.3/manual/index.html
[6] https://docs.unity3d.com/Packages/com.unity.inputsystem@1.7/manual/index.html
[7] https://visualstudio.microsoft.com/

4.2.2 Rider A powerful IDE from JetBrains designed for .NET development, offering advanced code analysis, debugging tools, and integration with Unity for efficient game development. Some of us are using Rider (see Fig. 27) as main IDE for programming C# scripts for Unity.[8]

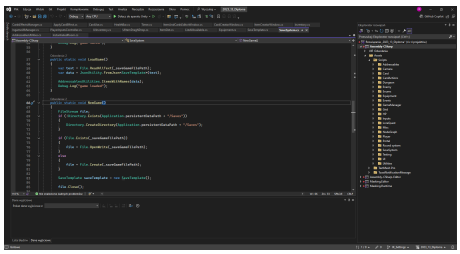


Fig. 26: Example of using Visual Studio by our team
Source: own elaboration

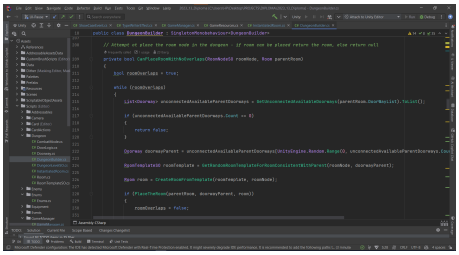


Fig. 27: Example of using Rider by our team
Source: own elaboration

4.3 Graphic Design Software

4.3.1 Pixel studio A lightweight, mobile-friendly app for creating pixel art and animations, aimed at artists looking to work on digital pixel graphics across devices. Someone of our team have used this program for one early-stage placeholder.[9]

[8] https://www.jetbrains.com/rider/
[9] https://store.steampowered.com/app/1204050/Pixel_Studio__pixel_art_editor/

4.3.2 Microsoft Paint A basic image editing tool that comes with Windows, providing simple drawing and painting functions suitable for quick, uncomplicated graphic tasks. It's our main tool in Graphic Design Software in the early stage of programming. Almost all graphical placeholders were made here (see Fig. 28)
(see Fig. 29).[10]

Fig. 28: Example of our old graphics made with "Microsoft Paint" *Source: own elaboration*

Fig. 29: Closer look into one item from the old graphics made with "Microsoft Paint" *Source: own elaboration*

[10] `https://www.microsoft.com/pl-pl/windows/paint`

4.3.3 Aseprite A dedicated pixel art and animation tool, popular among game artists for creating detailed sprite sheets, frame-by-frame animations, and retro-style graphics. It's our main Graphic Design Software (see Fig. 31). In our project almost all graphics and animation assets we created in this software (see Fig. 30).[11]

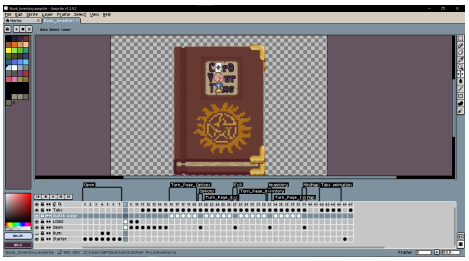


Fig. 30: Example of using "Aseprite" in creation inventory animation for our project
Source: own elaboration

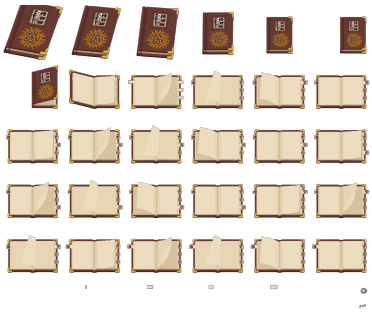


Fig. 31: Finished inventory animation exported by "Aseprite"
Source: own elaboration

4.4 Sound Design and Audio Editing Software

4.4.1 LabChirp A sound effect generator, primarily aimed at creating retro, 8-bit style audio effects commonly used in classic and indie games.Almost all SFX from the project were created in this beautiful software.[12]

[11] https://www.aseprite.org/
[12] https://labbed.itch.io/labchirp

4.4.2 sfxr Another sound effect generator designed for retro game sounds, enabling quick creation of lo-fi sound effects with minimal setup. Some of SFX from the project were created in this software.[13]

4.5 Digital Audio Workstation

4.5.1 DefleMask A tracker-based DAW specializing in chiptune and retro game music composition, supporting various old-school sound chip emulations. All music from the project were created in this software.[14]

4.6 Version control

4.6.1 Github A platform for version control and collaboration that uses Git, enabling developers to manage, share, and track changes in code projects. GitHub is the main version control system software for our Project (see Fig. 32)[15]

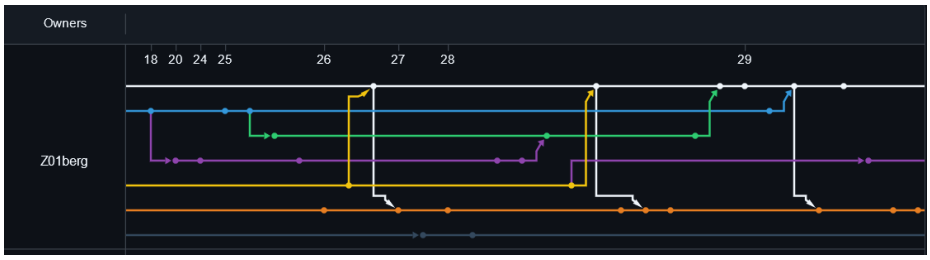


Fig. 32: Example of our team network graph presented by "Github" in label "Network"
Source: own elaboration

4.6.2 Jira An Atlassian project management tool used mainly for issue tracking and agile project management, common in software development workflows. Jira was our main task management software in the early stages of the project.[16]

[13] `https://www.drpetter.se/project_sfxr.html`
[14] `https://www.deflemask.com/`
[15] `https://github.com/`
[16] `https://www.atlassian.com/software/jira`

4.6.3 Trello A visual project management tool that uses boards, lists, and cards to organize tasks, widely used for managing project workflows. Trello is our current main task management (see Fig. 33) and backlog (see Fig. 34) software.[17]

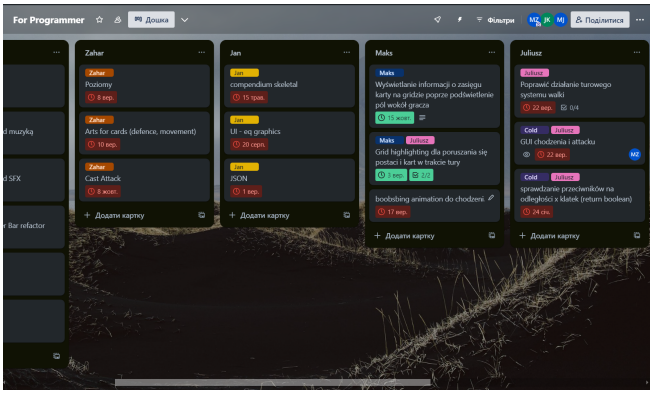


Fig. 33: Example of our tasks To-Do in Trello with deadlines, and tags
Source: own elaboration

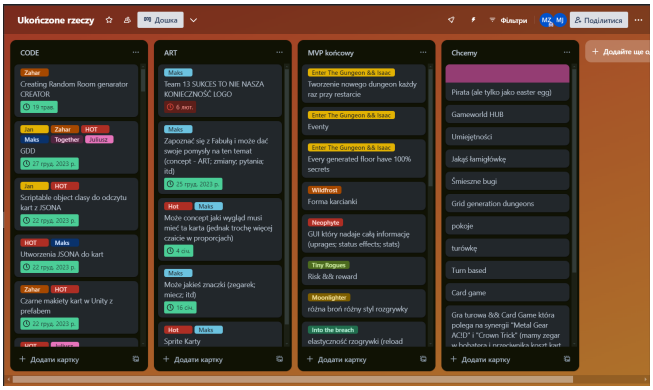


Fig. 34: Example of our backlog in Trello with done tasks and MVP GDD
Source: own elaboration

[17] `https://trello.com/pl`

4.7 Communication Platforms

4.7.1 MS Teams A collaboration platform by Micro soft, combining video meetings, chat, file sharing, and productivity tools for business and team communication. This platform we mainly used to class discuss tasks and if nobody responded on **Discord** because of the lecture, here respond will punch back immediately (see Fig. 35). The main channel of communication with our supervisor, reviewer and group information overall was through this platform (see Fig. 36).[18]

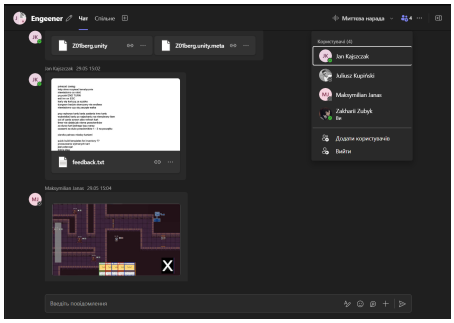


Fig. 35: Example of our private communication done via "Microsoft Teams"
Source: own elaboration

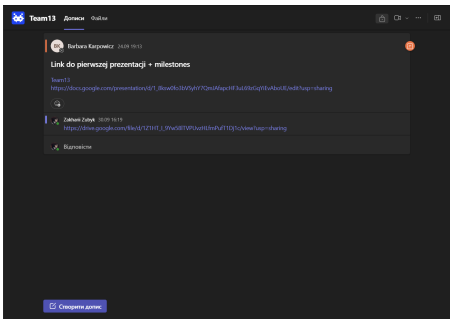


Fig. 36: Example of organization presentation of our reviewer done via "Microsoft Teams"
Source: own elaboration

[18] `https://www.microsoft.com/pl-pl/microsoft-teams/log-in`

4.7.2 Discord A versatile chat platform popular among gamers and communities, featuring voice, video, and text channels along with screen sharing and custom server options. Our main system of communication and information holder (see Fig. 37). Discord we used as well as Planner for meetings and platform for meetings it's self (see Fig. 38).[19]

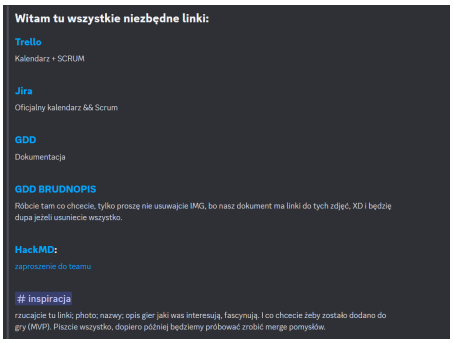


Fig. 37: Example of text channel with main information holder and links organized via "Discord"
Source: own elaboration

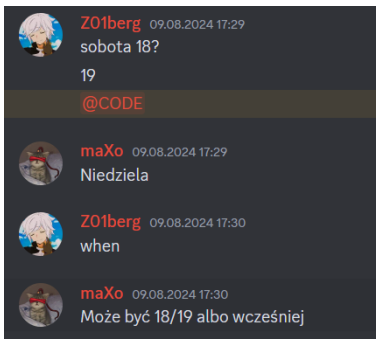


Fig. 38: Example of planning every week meeting of our team via "Discord"
Source: own elaboration

4.8 Documentation

4.8.1 Typora A markdown editor that provides a seamless WYSIWYG experience, making it easy to write, preview, and export markdown-based documents. Typora was our main documentation software in the early stages of the project. Our **early stage GDD**[20]

[19] `https://discord.com/`
[20] `https://typora.io/`

4.8.2 Overleaf An online collaborative LaTeX editor for writing, editing, and sharing complex documents, often used in academic and technical documentation. Overleaf is our current main documentation software (see Fig. 39).[21]

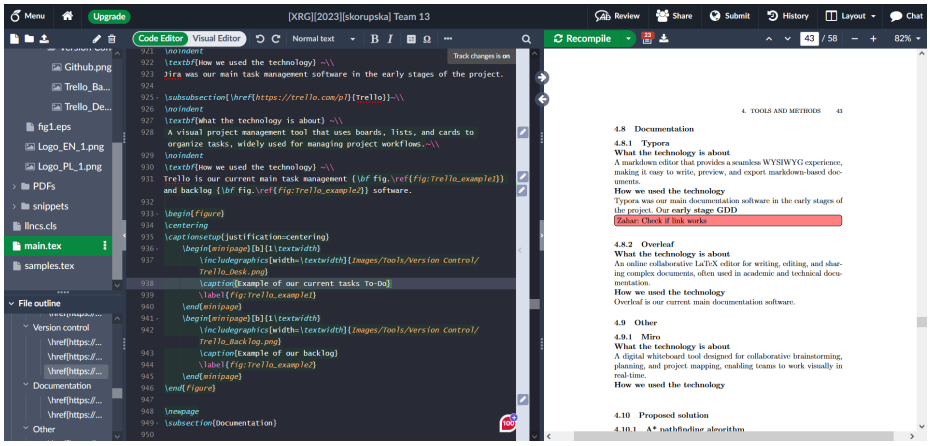


Fig. 39: Example of using Overleaf by our team
Source: own elaboration

4.9 Proposed Solutions

Remarkable note is that we are using those solutions in very different spaces and even if it was described only on one example of code, it doesn't mean that it's not used anywhere else. Examples were determined by clarity or simplicity only.

4.9.1 State Pattern: Game State

We are using Game State pattern to control properly Random Dungeon Builder and understand in which state of the game Player is. **Listing 85**

[21] `https://www.overleaf.com`

4.9.2 Singleton Pattern: Game Manager

We are using this pattern in different systems, but for this example I will provide Game Manager.cs. Because it combines two patterns inside.

We are using Singleton Pattern to operate the progress of the game with different states from **Listing 85**. It works like a queue with different tasks which singleton is handling only if state changed, but only one task at a time. It has full information of all levels, dungeons, player and state which is right now **Listing 81**.

4.9.3 Object Pooling Pattern: Dungeon Level SO

So in our game, we have. Random Dungeon Generator on ScriptableObjects and a build-in editor which uses prefabs of rooms **Listing 66** and graph system **Listing 65** to generate a full dungeon. Maybe it's not in the classic meaning of Pooling Pattern, because it's really reusing prefabs, but not the same objects. The only thing which it reuses without destroying is the main character. But still, it uses the main thesis of this Pattern and reuses the same prefabs just adjusting them with opening or closing pathways, so it's more reusable and feels as new every time **Listing 67**.

4.9.4 Observer Pattern: Health Bar

We are using Health Bar **Listing 88** only in pair with Timer. when the Timer is initialized, it has its ID and tag, so we should assign a Health bar to it to constantly check if this HP is zero and which object should die. Yeah, maybe it's not the best use of this pattern but the Timer and Health bar was created at the very beginning of our journey and too deep inside of our systems to rework.

4.9.5 Event Aggregator Pattern: Event System

We are using Event Aggregator Pattern in Event System **Listing 80**. We are centralizing events for the game and reducing direct dependencies among components.

4.10 Testing Protocols

4.10.1 Testing Environment: Test were conducted in 4 testing sessions taking part on April 10, May 8, May 29 and June 19. On these days build of the game with It's current progress was play tested by our teachers and colleagues. After each section we interviewed the player and received valuable feedback.

4.10.2 Testing Scenarios:
First testing session: Testers were testing: player movement, first prototype of equipment, card selection from hand, first grid implementation, turn system, and path finding.
Second testing session: Testers were testing: features mentioned in first testing session, main menu, pause menu, deck building, new grid implementation using card abilities, dealing damage to the enemies.
Third testing session: Testers were testing: features mentioned in previous testing sessions, random dungeon generation, healing the player, button rebinding, enemy behavior, first PVE combat and basic mini-map.
Fourth testing session: Testers were testing: features mentioned in previous testing sessions, deck management out of combat, card reset while in combat and game was beatable.

4.10.3 Test Cases

Main Menu
Environment: Main menu
Precondition: Turn on the game
Used in: Testing sessions 2-4

Step	Instructions	Expected Behavior
1.	Enter the game	Menu is visible
2.	Press on credits button	Credits screen opens
3.	Press on back button	Return to main view
4.	Press on settings button	Settings window opens
5.	Press keys button	Keys window opens
6.	Press settings button	Settings window opens
7.	Press back button	Return to main view
8.	Press start game button	Enter first scene

Table 1: Main menu test case

Pause menu
Environment: Game map
Precondition: Turn on the game, start the game
Used in: Testing sessions 2-4

Step	Instructions	Expected Behavior
1.	Press Esc	Pause menu opens
2.	Press Esc	Pause closes
3.	Press Esc	Pause menu opens
4.	Press main menu button	Exit to main menu

Table 2: Pause menu test case

Player movement
Environment: Game map
Precondition: Load into the game, create your deck by choosing your equipment
Used in: Testing sessions 1-4

Step	Instructions	Expected Behavior
1.	Press Enter	Game started counting down numbers
2.	press A button / L arrow	Player step left 1 grid
3.	press D button / R Arrow	Player step right 1 grid
4.	press W button / U Arrow	Player step up 1 grid
5.	press S button / D Arrow	Player step down 1 grid

Table 3: Player movement test case

Deck building
Environment: Equipment
Precondition: Load into the game
Used in: Testing sessions 3-4

Step	Instructions	Expected Behavior
1.	press E	Opened eq window with default cards
2.	Using LMB press and hold on item and drag it	Item snaps to mouse position
3.	Releasing LMB drop item outside of inventory.	Item goes back to its starting position
4.	Repeat step 3. By releasing LMB drop an item on the wrong EQ slot.	Item goes back to its starting position
5.	Repeat step 3. By releasing LMB drag item to the correct slot in EQ.	
6.	Drag other items to the right slots	All cards change
7.	Press E	EQ closes, saves cards and creates your deck

Table 4: Deck building test case

Card abilities
Environment: Game map
Precondition: Load into the game and create your deck
Used in: Testing sessions 2-4

Step	Instructions	Expected Behavior
1.	Hover any card in your hand by putting your mouse cursor over it.	Card gets bigger and comes to the front of all cards
2.	Use any card by pressing on it with LMB	Card should come up to the front and go up
3.	Cancel your choice by pressing on the previously highlighted card with RMB	Card should go back to it's previous position
4.	Choose your card again by repeating step 2	
5.	Attack the enemy with the card of your choice by pressing on him with LMB	Enemy should receive damage or healing depending on the cards stats, damage should pop up over enemy's head, card cost should be added to player's timer, card should be destroyed
6.	Finish your turn by pressing Enter	Your turn should finish and next character on the map should make their action

Table 5: Card abilities test case

Round System
Environment: Game map
Precondition: Start the game
Used in: Testing sessions 1-4

Step	Instructions	Expected Behavior
1.	Go to enemy using instructions from the Test Case 01	
2.	Press Right CTRL	
3.	Press ” (quotation mark)	Your timer should increase by 1
4.	Press / (slash)	Your timer should decrease by 1
5.	Hold ” (quotation mark) until your timer will not be higher than enemy timer	
6.	Press ENTER	Timers should start counting down and Enemy turn should start
7.	Using WASD or LRUD ARROW	Player shouldn’t move
8.	Press , (comma) or . (dot)	
9.	Press ” (quotation mark) until enemy timer will be higher than the Player’s timer	
10.	Press ENTER	Timers should start counting down and Player turn should start

Table 6: Round System test case

Enemy path finding abilities
Environment: Game map
Precondition: Turn on the game and choose scene 2 in the main menu.
Used in: Testing sessions 1-4

Step	Instructions	Expected Behavior
1.	Move the Mouse Cursor on a chosen tile.	The walkable tile is marked black, the unusable tile stays the same
2.	Press the Left Mouse Button on a chosen tile.	If possible, the enemy moves to the selected tile.
3.	Press C	The enemy moves to the player's current position.
4.	Press V	The enemy starts chasing the player continuously.
5.	Press V while the enemy is chasing the player	The enemy moves to the player's last known position and stops.
6.	Press I	The enemy's movement speed is increased
7.	Press P	The enemy's movement speed is decreased

Table 7: Enemy path finding test case

5 Results

This section explains how game systems were implemented, using examples of code, and descriptions of tools used to achieve each of the listed Systems.

5.1 Inventory

The inventory system is one of the first systems implemented in the game. It consists of many scripts and windows, that let the player create his deck of cards from the items collected during his adventures in the dungeon. This subsection explains how this system has been implemented.

5.1.1 Overview

Starting with the basics of the system. All items currently owned by the player are stored in the "Inventory" class. All items equipped by him are also stored in the "Equipment" class. Both of these classes implement the "Singleton" design pattern, which gives us much easier access to them. The "Inventory" class holds the items in two separate lists, one is for storing them long term, and the other holds them until the inventory is opened and items are added to the items window. The "Equipment" class holds all items chosen by the player, assigned to the corresponding places, hand or head can be taken as an example.

When the "E" key is pressed the inventory window opens. The script will then list all items that are in the "Inventory" class but have not yet been displayed. The player then sees three panels, one holds all not-equipped items, another all equipped items together with places they were put in, and the last one holds all cards that will end up in the player's deck when the window is closed
(**see Fig. 40**).

Fig. 40: Windows of the inventory.
Source: own elaboration

Once the window is opened, the player can choose items, by dragging them or double-pressing, and placing them into slots inside the equipment panel. Additionally when a player hovers over an item slot all items and cards that are, or can be tied to it are highlighted. Once the item is placed, cards in the cards window corresponding to the chosen item slot are switched with the cards coming from the chosen item. Every item slot in the equipment panel has a set of default cards that are listed in the cards panel until an item is put into that slot.

At the end, the window can be closed. At that point, all items chosen by the player are saved to the "Equipment" class, and a deck of cards is created, from cards inside the chosen items, together with leftover default cards.

5.1.2 Item Slots

Item slots are places where items that belong to the player can be put. Each item has its own slot in the items panel (**see Fig. 41**). There are also several slots from which the equipment panel is built.

Each item slot has been given a type of item that can be placed into it.

Fig. 41: Example of an item slot.
Source: own elaboration

The slots consist of many mechanics, but for now, we will focus on the basics alone. Each slot inside the items panel is added when the window opens and there are items that have not yet been displayed. Each of the slots also contains a basic script, but the slots inside the equipment panel also contain a script that manages the listing and delisting of default cards in the cards panel.

5.1.3 Highlighting of Items and Cards

One of the mechanics implemented in the item slot script is the highlighting of cards and items once a player hovers over a chosen slot (**see Fig. 42**). The script only highlights items that fit the type of the chosen item slot. If the slot is positioned inside the equipment panel, highlighted are also cards that are tied to that slot or item inside it. Hovering over a slot in the items panel will only highlight slots inside the equipment panel.

Fig. 42: Highlighted items and cards in inventory window
Source: own elaboration

The basic concept of the system is quite simple. First, each item slot has to be added to a common static list when created. It is done with the use of Unity's built-in "OnEnable()" and "OnDisable()" methods (**Listing 1**). Then each time the player's mouse enters the bounds of the item slot object, a function is called (**Listing 2**) that will look for item slots containing items fitting the item type of the chosen slot. In all slots found, the color is changed to be slightly green.

```
1 private void OnEnable()
2 {
3     allItemSlots.Add(this);
4 }
5 private void OnDisable()
6 {
7     allItemSlots.Remove(this);
8 }
```

Listing 1: OnEnable and OnDisable Methods
Source: own elaboration

The method will also check if each item slot is inside the items panel, and if so, it will only highlight it if the player is currently hovering over a slot inside the equipment panel. For this purpose, each slot can be of item type "any".

```
if (transform.childCount == 0) return;
ItemType childType = GetComponentInChildren<UIItemDragNDrop>().item.
    itemType;
foreach (ItemSlot slot in allItemSlots)
{
    if (slot == this) continue;
    if (slot.allowedItemType == childType)
    {
        if (command == "lightup")
        {
            slot.LightUp();
        }
        else
        {
            slot.LightDown();
        }
    }
}
```

Listing 2: Example of searching through the slots.
Source: own elaboration

The method will also look through all cards inside the item slot (**Listing 3**), or item inside a chosen slot, and also highlight them. The cards will be placed at the top of the cards panel by changing their position in the scene hierarchy to be the first child of a parent game object.

```
foreach (var card in GetComponent<DefaultCards>()._cardsList){
    if (command == "lightup"){
        card.transform.Find("ArtworkMask").transform.Find("BG").
    GetComponent<Image>().color = _mouseOverColor;
        card.transform.SetAsFirstSibling();
    }else{
        card.transform.Find("ArtworkMask").transform.Find("BG").
    GetComponent<Image>().color = Color.white;
    }
}
```

Listing 3: Example of highlighting cards.
Source: own elaboration

5.1.4 Moving Items

There are two ways to move items between slots in the game. Players can either drag the item to a chosen location or double-click on it. First, let us take a look at the dragging mechanic.

Players can initialize drag by pressing and holding the left mouse button over an item. From that point, the position of the item on the screen will be the same as the mouse position.

When a player releases the button, two possible things can happen. If the item is not dropped into a slot inside the equipment window, it will go back to its original parent slot inside the items panel. Otherwise, if the item is dropped on a slot in the equipment panel, a function is called inside the slots script (**Listing 4**) that will check if the item type of slot and the item match, and if it does, change the parent object of the item and sets its position to fit the chosen slots position. While the item is being dragged, its parent is changed to an object higher in the scene hierarchy to ensure that it is always displayed in front of the inventory windows. Once the item is dropped, its parent will change to the parent it was taken from.

```
1  public void OnDrop(PointerEventData eventData)
2  {
3      if (eventData.pointerDrag != null && eventData.pointerDrag.
       GetComponent<UIItemDragNDrop>().item.itemType == allowedItemType &&
       transform.childCount == 0)
4      {
5          eventData.pointerDrag.GetComponent<UIItemDragNDrop>().
       BackToTempParent();
6          eventData.pointerDrag.GetComponent<UIItemDragNDrop>().
       _tempOldParent = gameObject;
7          eventData.pointerDrag.transform.SetParent(transform, false);
8          eventData.pointerDrag.GetComponent<RectTransform>().position =
       transform.GetComponent<RectTransform>().position;
9      }
10 }
```

Listing 4: OnDrop item slot method.
Source: own elaboration

Another way of moving items is by double-pressing on them (**Listing 5**). When the player double clicks on the chosen item, a raycaster first checks if the pressed game object is an item slot, then

a function is called on the item slot that will change the parent of the item inside to the first slot inside the equipment panel that fits the item type. If the item slot pressed is inside the equipment panel, the function will change the parent of the item to its original parent inside the items panel.

```
public void OnDoubleClick(InputAction.CallbackContext context)
{
    List<RaycastResult> results = new List<RaycastResult>();
    PointerEventData d = new PointerEventData(UnityEngine.EventSystems.
    EventSystem.current);
    d.position = Mouse.current.position.ReadValue();
    _raycaster.Raycast(d, results);
    foreach (RaycastResult r in results)
    {
        if(r.gameObject.CompareTag("ItemSlot"))
        {
            r.gameObject.GetComponent<ItemSlot>().DoubleClicked();
            break;
        }
    }
}
```

Listing 5: OnDoubleClick method
Source: own elaboration

Both ways of moving items are implemented partially inside the item's script and the item slot. Raycasting is however implemented in the player's inputs controller script.

5.1.5 Default Cards

Each item slot inside the equipment window contains a set of cards that can be part of the player's deck if the slot to which they are tied remains empty when the player closes the equipment window. These cards will ensure, that even without any or most of the item slots taken, the deck will still consist of enough cards to make the game playable.

The way the script works is quite simple. The "Update()" method continuously checks if an item slot is empty. If it is and the cards have not yet been displayed, the "DisplayCards()" method will create a card game object for each of cards scriptable objects contained inside a list. These scriptable objects are given to the item slot by the use

of the "Addressable" system when the level loads in, imported cards are tagged as "DefaultCard". All cards created by the method will be added to the cards panel, and also to a cards list inside the script.

If the "Update()" method notices an item inside the slot and there are default cards still inside the cards list, the "HideCards()" method is called that destroys all game objects inside the cards list, which will remove them from the cards panel.

5.1.6 Listing Items

The items are added to the inventory window each time it is being opened. The inventory consists of two lists, one with all items in the player's possession, and one only with items that have not yet been displayed in the inventory window. Once the window opens the script will go through all the items not yet displayed and create necessary game objects for each scriptable object inside the list. The game objects consist of the item game object and item slot. The parent of the item is set to be the item slot and the slots parent is set to be the items panel. Lastly, the script checks if the item scriptable object should be put inside any place in the equipment (**Listing 6**) and if it should, the script will call a method inside the item slot called "DoubleClicked()" that will move the item to a first found empty slot inside the equipment panel that fits the type of an item.

```
var eq = Equipment.Instance;
if ((eq.leftHand != null && eq.leftHand.itemName == x.itemName) ||
    (eq.rightHand != null && eq.rightHand.itemName == x.itemName) ||
    (eq.boots != null && eq.boots.itemName == x.itemName) ||
    (eq.legs != null && eq.legs.itemName == x.itemName) ||
    (eq.chest != null && eq.chest.itemName == x.itemName) ||
    (eq.head != null && eq.head.itemName == x.itemName) ||
    (eq.item1 != null && eq.item1.itemName == x.itemName) ||
    (eq.item2 != null && eq.item2.itemName == x.itemName) ||
    (eq.item3 != null && eq.item3.itemName == x.itemName) ||
    (eq.item4 != null && eq.item4.itemName == x.itemName) ||
    (eq.item5 != null && eq.item5.itemName == x.itemName) ||
    (eq.item6 != null && eq.item6.itemName == x.itemName))
{
    slot.GetComponent<ItemSlot>().DoubleClicked();
}
```

Listing 6: Checking if item should be in equipment.
Source: own elaboration

5.1.7 Listing Cards

The cards are added to the cards panel or removed from it any time an item is either inserted or removed from the equipment. Each item holds in itself two lists, one with card scriptable objects, and the other, holding card game objects that have been created inside the cards panel. The logic of the system mostly consists of three methods, the most important of which is the "OnItemChangePlace()" (**Listing 7**).

```
public void OnItemChangePlace(bool doubleClicked = false)
{
    canvasGroup.alpha = 1f;
    canvasGroup.blocksRaycasts = true;
    BackToTempParent(doubleClicked);
    if (transform.parent.transform == parentTransform)
    {
        if (originParentTransform == null)
        {
            var sl = Instantiate(itemSlotPF);
            sl.GetComponent<ItemSlot>().allowedItemType = ItemType.any;
            sl.transform.SetParent(itemsPanel.transform);
            originParentTransform = sl.transform;
        }
        rectTransform.position = originParentTransform.position;
        rectTransform.SetParent(originParentTransform);
        RemoveCardsFromDeck();
    }
    else
    {
        AddCardsToDeck();
    }
    parentTransform = transform.parent.transform;
}
```

Listing 7: OnItemChangePlace Method
Source: own elaboration

The method will first check if the item's parent game object has changed. If it did, then the script will call a method that will instantiate card game objects for each card scriptable object tied to the item. The script will also remove scripts from the card that are not needed while the card is inside the inventory window. If however, the parent item did not change when the item was dropped, it means that the item was dropped outside of the bounds of any slot, or that it was dropped on a slot inside the items panel since the slots of

the items panel will early return the method called when an item is dropped. This makes it easier to control when to display and hide the cards since we only have to check if items parent changed at all. So if the parent has not been changed and the item was dropped, the item must be reassigned to the original parent and the card game objects inside the cards list have to be destroyed. In this case, the method will also check if the item has an item slot already assigned as an original parent, and if not, a new slot will be created.

5.1.8 Saving equipment state

The state of the equipment will save any time the inventory window is closed. The "SaveState()" method is called (**Listing 8**), which will first clear the deck and then go through all the cards in the cards panel and add all the cards scriptable objects from all of the cards. Then it will check each of the item slots in the equipment panel and add an item scriptable object from the item inside the slot to its place in the equipment script. If there is no item in the slot, the null value is saved.

```
1  public void SaveState()
2  {
3      Equipment.Instance.cards.Clear();
4      foreach (Transform card in _cardsPanel.transform)
5      {
6          Equipment.Instance.cards.Add(card.GetComponent<CardDisplay>().
       cardSO);
7      }
8      _deckController.ManageDeck();
9      GameObject slot = _equipmentPanel.gameObject.transform.Find("HeadSlot"
       ).gameObject;
10     if(slot.GetComponentInChildren<UIItemDragNDrop>() != null)
11     {
12         Equipment.Instance.head = slot.GetComponentInChildren<
       UIItemDragNDrop>().item;
13     }
14     else
15     {
16         Equipment.Instance.head = null;
17     }
```

Listing 8: SaveState equipment Method
Source: own elaboration

5.2 Items and Cards

All items and cards are implemented as scriptable objects. These objects hold information needed to modify game objects to which they are tied. A card or item game object can be created, and each of them has a variable in which a scriptable object can be put. When the game object is instantiated the script will take information inside the scriptable object and modify the game object accordingly, for example giving it a name, a description, or a graphic and a background.

5.3 Cards and Items Creator

"Item and cards creator" is a tool used for creating and balancing items and cards in our game. The program can also import cards and items from json files. The tool can be accessed at the top menu of Unity engine window (**see Fig. 43**).

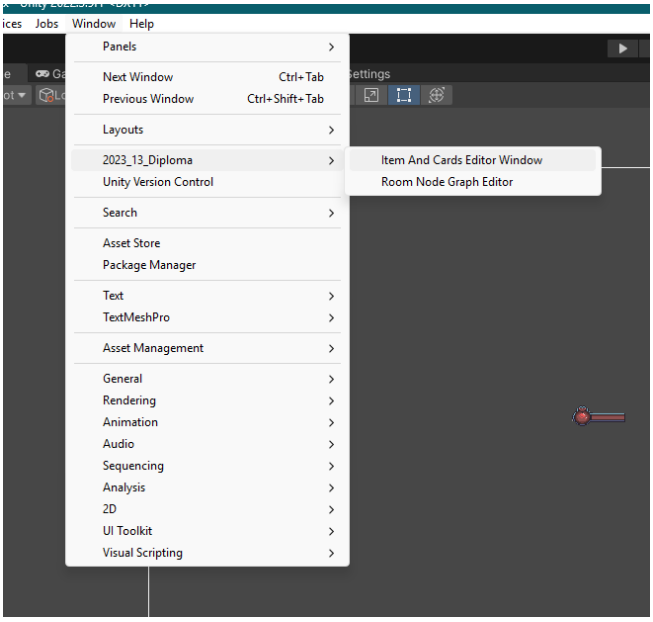


Fig. 43: Creator in the window menu.
Source: own elaboration

After opening the tool, a window appears(**see Fig. 44**).

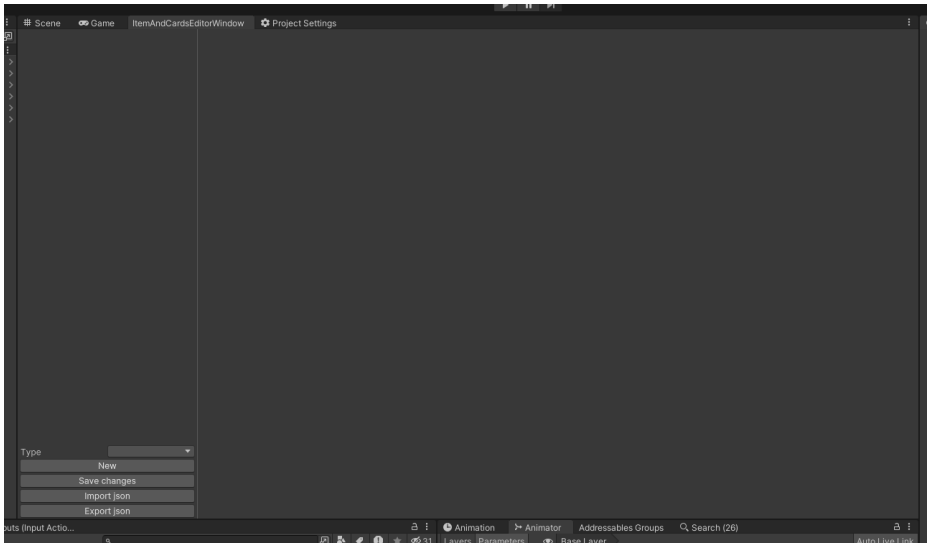


Fig. 44: Creator window.
Source: own elaboration

On the bottom left side, several buttons and a drop-down menu with "Type" label have been positioned. The user has two options to choose from. He can either display a list of cards or a list of items. When a user chooses an option from a list a creator appears on the right side of the window (**see Fig. 45 and Fig. 46**).

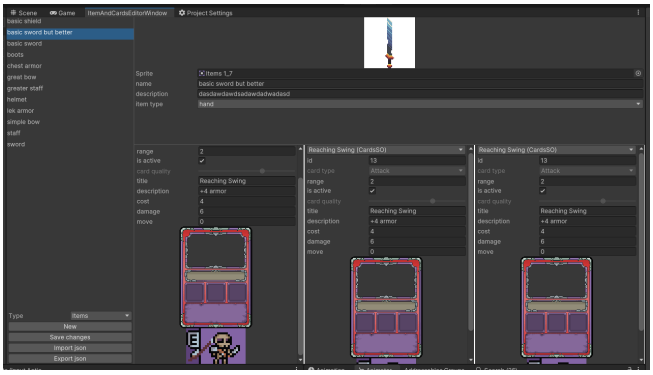


Fig. 45: Item editor window.
Source: own elaboration

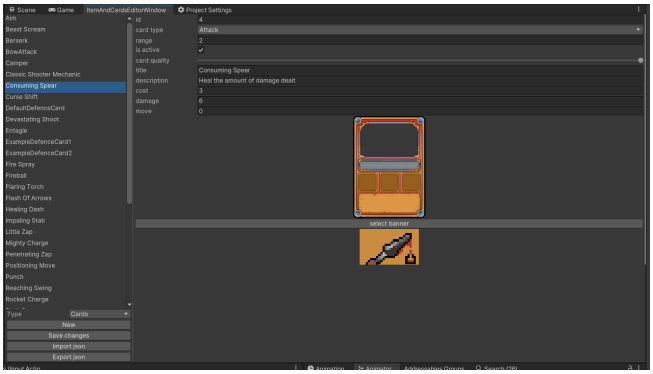


Fig. 46: Card editor window.
Source: own elaboration

When the “New” button is pressed, the same windows appear, only without data, and not tied to any existing object. Buttons “Import json” and “Export json” save and load cards and items from outside files. “Save” button saves changes made to objects currently displayed in the right window. The Main window is created in the class “ItemAndCardsEditorWindow.cs” inheriting from the class “EditorWindow”.

The creator systems script starts in the “ItemAndCardsEditorWindow” class. When the editor is opened the first method called is “ShowWindow()”, it sets the default size of the window and gives it a name. At last, it calls the “CreateGUI()” method.

The “CreateGUI()” function is where things get a bit more complicated. It divides all parts of the whole editor window into parts. It also initializes all buttons and gives values to all the popup fields and other similar controls.

The asset being edited in the editor can change at any time, tho if not saved the asset changes will not remain. That functionality is implemented in the “OnItemSelectionChange()” method (**Listing 9**). It follows what item in the list is currently selected and creates windows on the right side. The method will further divide the windows to fit as many elements as each object needs. For example, for editing cards, we only need one window, but for editing items, we need a sum of four windows, one for the item itself, and three for the cards inside the item.

The function first starts by clearing the editor window from all elements, then it checks if there is in fact a new object selected. If there is one, the function will move forward.

```
1  private void OnItemSelectionChange(IEnumerable<object> selectedItems)
2  {
3      m_RightPane.Clear();
4      _rightPaneWindows.Clear();
5      var enumerator = selectedItems.GetEnumerator();
6      if (enumerator.MoveNext())
7      {
8          var spriteImage = new Image();
9          spriteImage.scaleMode = ScaleMode.ScaleToFit;
10     }
```

Listing 9: OnItemSelectionChange Method
Source: own elaboration

Next, there are two ways the method can go. If the chosen object is an item (**Listing 10**), the method will take different values from the object, and then by the use of a constructor an item creator window will be created, and a card creator window will be created for each card inside the item. The card creator windows will be positioned at the bottom of the right side window in a row.

```
1       if (_selectedAdressableType == "Items")
2       {
3           var splitViewCards = new TwoPaneSplitView(0, 250,
    TwoPaneSplitViewOrientation.Vertical);
4           m_RightPane.Add(splitViewCards);
5           VisualElement lower = new VisualElement();
6
7           var selectedItem = enumerator.Current as Item;
8           if (selectedItem == null) return;
9
10          spriteImage.sprite = selectedItem.icon;
11          string name = selectedItem.itemName;
12          string desc = selectedItem.description;
13
14          List<CardsSO> cards = new List<CardsSO>(selectedItem.cards);
15          var itemCreatorWindow = new ItemCreatorWindow(selectedItem,
    name, desc, selectedItem.itemType, spriteImage.sprite);
16
17          splitViewCards.Add(itemCreatorWindow);
18          _rightPaneWindows.Add(itemCreatorWindow);
19
20          splitViewCards.Add(lower);
21
22          foreach(var card in cards)
23          {
24              lower.style.flexDirection = FlexDirection.Row;
25              lower.style.justifyContent = Justify.Center;
26
27              var cardCreatorWindow = new CardCreatorWindow(card,
    selectedItem);
28              lower.Add(cardCreatorWindow);
29              _rightPaneWindows.Add(cardCreatorWindow);
30          }
31      }
```

Listing 10: OnItemSelectionChange Method if item selected
Source: own elaboration

Otherwise, if a chosen object is a card, the creator will only create a single window (**Listing 11**), since only the chosen card needs to be displayed Differently from the item creator, to create a card creator

we can give the constructor the whole object instead of dividing it into variables.

```
        else if(_selectedAdressableType == "Cards")
        {
            var selectedItem = enumerator.Current as CardsSO;
            if (selectedItem == null) return;
            var cardCreatorWindow = new CardCreatorWindow(selectedItem);
            m_RightPane.Add(cardCreatorWindow);
            _rightPaneWindows.Add(cardCreatorWindow);
        }
    }
}
```

Listing 11: OnItemSelectionChange Method if card selected
Source: own elaboration

To keep track of what asset types should be displayed in the assets list the "OnAddressablesSelectionChange()" is used. It is attached to the asset selection popup field. The main functionality of the method is quite simple. When it is called it will call the AddressablesUtilities.LoadItems method which returns a list of objects that then have to be converted to a desired object type (either "CardSO" or "Item". Then the list of assets is cleared and filled with new ones. Lastly, the assets displayed are switched to the newly imported ones.

New items and cards can be created by pressing the "New" button. The process of creating the windows for it goes similarly to selecting an already existing object, the only difference being that here we do not need to pass any data to the cards and items windows constructors.

The last methods of class "ItemAndCardsCreatorWindow", are "ImportJson()", "ExportJson()", and "Save()". Importing json files is being done in the Deserialization.cs class. Exporting objects to json is as of now not needed, so the function is here in case it would be in the future.

The Save function works by iterating over a list of windows currently in the right pane of the main window, and calling the "Save()" method in each of them. All of those windows implement the IModifiable interface.

The next part of the creator window is the ability to import assets from json files. For this functionality, several additional classes had to be created, mirroring the json representations of cards and items. Each card type has been put in its json file.

The class responsible for the whole import process is called “Deserialization.cs”. It holds data necessary for the import to take place. That is: Json file reference, Json objects list, and a path to a folder where the assets will be created. It holds that data for each type of object it will be importing.

The process of Importing is started by calling an ”Import()” public function.

The process for card import looks the same for each card type but had to be done 4 times because of some naming differences.

```
objects = JsonUtility.FromJson<JsonAttackCards>(AttackFile.text);

foreach (var obj in objects.attackCardsList)
{
    var s = ScriptableObject.CreateInstance<CardsSO>();

    s.id = obj.id;
    s.type = CardType.Attack;
    s.isActive = obj.isActive;
    s.cardQuality = obj.cardQuality;
    s.title = obj.title;
    s.description = obj.description;
    s.cost = obj.cost;
    s.damage = obj.damage;
    s.move = obj.move;
    s.backgroundPath = "Graphics/CardBackgrounds/AttackCards/Attack" +
    PickCardQuality(s.cardQuality);
    s.spritePath = $"Graphics/CardSprites/AttackCards/{s.title}";
    s.range = obj.range;

    AssetDatabase.CreateAsset(s, cPath + s.title + ".asset");
    AssetDatabase.SaveAssets();

    AssignAsAddressable(s, "AttackCardsGroup", "AttackCard",  s.
    cardQuality == 0);
}
```

Listing 12: Importing cards from json
Source: own elaboration

The above part of “Import()” method (**Listing 12**) can be used as a representation. The list of json cards is being loaded, via the built-in “JsonUtility” class. The list is then iterated through and for each object in it, a new CardsSO object is created and the data is assigned to the new objects variables.

The paths for the card graphics can be mostly predefined since they will not change. The only part that needs to be added is the name of the card or card quality. Then the asset is created and saved via “AssetDatabase” built-in class. At the end, the object is assigned as an addressable asset and added to the addressable registry.

The process of importing items is similar to cards (**Listing 13**). There are however two differences. Each item has been assigned a place in equipment it can be assigned to. When importing, those have to be converted from strings to enum ItemType.

The item object must next be assigned a list of cards it holds inside itself. Since the importing is not done during the game but rather in the unity editor, “AssetDatabase” class can be used for this purpose. First, a Guid for a searched card asset is imported, then it is converted to a path, and lastly, an asset is loaded from the path.

At last, an item asset is created and assigned as addressable with the correct tags given.

```
itemsObjects = JsonUtility.FromJson<JsonItems>(jsonItemsFile.text);
foreach (var obj in itemsObjects.itemsList)
{
    var s = ScriptableObject.CreateInstance<Item>();
    switch (obj.itemType)
    {
        case "hand":
            s.itemType = ItemType.hand;
            break;
        case "cheast":
        .
        .
        .
        case "any":
            s.itemType = ItemType.any;
            break;
    }
    s.itemName = obj.itemName;
    s.description = obj.description;
    foreach(var c in obj.cards)
    {
        var guids = AssetDatabase.FindAssets("t:" + typeof(CardsSO),null);
```

```
23
24          foreach(var guid in guids)
25          {
26              string p = AssetDatabase.GUIDToAssetPath(guid);
27
28              if (Path.GetFileNameWithoutExtension(p) == c && p != null)
29              {
30                  s.cards.Add(AssetDatabase.LoadAssetAtPath<CardsSO>(p));
31              }
32          }
33      }
34      s.icon = Resources.Load<Sprite>(obj.icon);
35      AssetDatabase.CreateAsset(s, itemsPath + s.itemName + ".asset");
36      AssignAsAddressable(s, "Items", "Item", false);
37      AssetDatabase.SaveAssets();
38  }
```

Listing 13: Importing items from json
Source: own elaboration

Since the quality of a card is saved as an integer, and the background is named using Roman numerals, a conversion has to happen.

Both 0 and 1 quality cards are assigned with "I". It is necessary because 0 quality cards are assigned as default and appear in the player's inventory when no item has been chosen for a slot but they still use graphics of quality 1. The method uses a simple switch statement.

All objects can be assigned as addressable via a method (**Listing 14**). The method uses Addressables built-in functions to put a chosen object in the registry, assign it to a group, give it a tag, and check if the object will be used as a default object, for example, a default card. An asset can be easily imported while in the Unity editor by first finding the asset file path, then the path can be converted into Guid. Then an addressable asset is created or edited, with Guid and a group name. Lastly, the asset has to be tagged properly.

```
1  private void AssignAsAddressable(Object asset, string targetGroup, string
       targetLabel, bool isDefault)
2  {
3      AddressableAssetSettings settings =
       AddressableAssetSettingsDefaultObject.Settings;
4      string assetPath = AssetDatabase.GetAssetPath(asset);
5      string assetGUID = AssetDatabase.AssetPathToGUID(assetPath);
6      var group = settings.FindGroup(targetGroup);
7      var entry = settings.CreateOrMoveEntry(assetGUID, group);
8      entry.SetLabel(targetLabel, true, true, true);
```

```
9
10      if (isDefault)
11      {
12          entry.SetLabel("DefaultCard", true, true, true);
13      }
14  }
```

Listing 14: Deserialization AssignAsAddressable Method
Source: Unity forum.[1]

The next part of the system is the "Item editor" window. The item editor window is created using the "ItemCreatorWindow" class inheriting from "VisualElement" and implementing "IModifiable" interface.

The created object holds references to fields that are created inside, a reference to an item object, and a path to where the item assets are held.

The class has two constructors. One remains empty for new item creation, and the second one puts data into window fields and assigns a reference to the item being edited.
At the start of object construction a "CreateFields()" method is called that instantiates all fields used in this window without putting data into it. Data is being assigned by another method.

The item can be assigned as addressable by using the same function as in Deserialization called AssigneAsAddressable().

The graphic for the item can be imported into the game via "ImportGraphic" button and method. The method has not yet been fully implemented here.

Changes done to an Item asset can be saved by using the Save() function of IModifiable interface. In this function, all data from the windows fields is assigned to asset variables. Item has to be given a name in order to be saved. If there is no item reference already, a new asset is created and assigned as addressable.

The last part of this system is the "Card editor" window. It has been constructed in a similar way to an item window. It uses A CardCreatorWindow class inheriting from the "ScrollView" class and implementing the "IModifiable" interface.

[1] https://discussions.unity.com/t/set-addressable-via-c/741902/13

Similarly to the item window, the object holds references to its fields and a card asset reference. Additionally, two Images are stored for the background and banner of a card. Also stored, are the current file paths to those graphics.

The object also keeps a reference to a saved card and an item the currently chosen card is assigned to if the editor has been created as part of the item creator.

The CardCreator class works mostly the same as the ItemCreator. The class contains two constructors, one for the new card and one for the already existing one. Differently from the item window the card window creates a new card in the constructor and gives it graphics file paths. The asset is not yet saved in the asset database, therefore if the user does not press the "Save" button, the object will be disposed of once the window is closed.

The "CreateFields()" method works the same way as in the item window, The difference being, that a popup field is created if the item reference exists, letting the user change cards assigned to the item. The change in card selection is being watched by the assigned "CardSelectionChanged()" method, which changes a card asset reference to a new one when a new card is chosen. The saved card reference stays the same and is changed only after "Save()" function is called. At the end "PopulateFields()" method populates fields of the window with the data from the newly chosen card.

When the card's name is changed a "CardNameChanged(ChangeEvent evt = null)" method is called. The method will automatically look for a banner graphic with the same name as the card's name. The function has to look inside different folders depending on what is the card type. The banner is automatically assigned to necessary fields, but not yet to the card asset itself. That will happen when the Save() method is called.

When either card type or card quality are changed a method called CardTypeChanged() is called. The method changes the values in the graphics path to match the folder names. For example, the attack cards are in the folder AttackCards and have the name "Attack" + quality of a card determined by a PickCardQuality() method, like in Deserialization. At the end a new background path is set as currently used, and the "CardNameChanged()" function is

called to find and assign a new banner of the card.

The class also consists of methods: “AssignAsAddressable()” which works the same way as in the item creator window, and “PopulateFields()” which works by assigning data to all fields defined by “CreateFields()”.

The user can choose a new banner via a button above the banner graphic in the editor window. The button opens a new window, letting the user pick a file. The “PickCardBanner()” method (**Listing 15**) is responsible for this mechanic. Unity has a builtin method, that returns a global path to a file chosen in a file explorer window. Once the file explorer window has been closed, the algorithm will check if the path is not empty. Next, the path needs to be converted into a local path. The extension of the chosen file also has to be removed from the path since the “Resources” system will not accept it otherwise. Lastly, a chosen graphic is assigned as a banner of the edited card. The method also changes the name of the card to the name of the graphic chosen.

```
private void PickCardBanner(){
    var file = EditorUtility.OpenFilePanel("graphic selection", "Assets\\
    Resources\\Graphics\\CardSprites", "");
    if (file == null || file.Length == 0 || file == "")
        return;
    StringBuilder sb = new StringBuilder();
    string[] folders = file.Split("/");
    var index = 0;
    _titleField.value = Path.ChangeExtension(folders[folders.Length - 1],
    null);
    for (int i = 0; i < folders.Length; i++)
    {
        if (folders[i] == "Graphics")
        {
            index = i;
            break;
        }
    }
    var path = String.Join("/", folders.Skip(index));
    path = Path.ChangeExtension(path, null);
    _banner.sprite = Resources.Load<Sprite>(path);
    _currentBannerPath = path;
}
```

Listing 15: PickCardBanner Method
Source: own elaboration

Saving a card is a bit more complicated than saving an item. Firstly if an item reference exists the cards in that item are switched to the currently chosen cards. Then all data is assigned from the window's fields to the card reference object. If the saved card reference is null, it means that a new card asset has to be created. The function will automatically pick an ID for the card out of all cards of the same type. And will also assign the newly created asset as addressable. At the end of "Save()" method the saved card referenced is assigned an object and the object is set as dirty, so it will be saved by the unity editor.

5.4 Main and Pause Menu

Every game, no matter how small or big, needs a reliable set of menus. In this game there also exists a set of two menus: "Pause" and "Main" menu.

First, let us start by explaining the basics of the "Main" menu system. The menu is built as a separate scene and is the first thing the player will see when entering the game. When the menu opens, the first thing a player can see is the main screen (**see Fig. 47**), on which a set of buttons is visible together with the game title. The continue button will only be visible if a save file exists.

Fig. 47: Main view of the Main menu
Source: own elaboration

In order to make sure that no windows are open at the start of the game, the menu script will use the "SetActive()" method on all of them (**Listing 16**), and set the activity to false. All methods concerning the main menu functionality are implemented in the "MainMenuManager.cs" class.

```
private void Start()
{
  if (!SaveSystem.CheckIfSaveExists())
  {
      _continueButton.SetActive(false);
  }
    _mainMenuView.SetActive(true);
    _settingsView.SetActive(false);
    _creditsView.SetActive(false);
    _compendiumView.SetActive(false);
}
```

Listing 16: Main menu Start Method
Source: own elaboration

Most buttons, though opening different windows, work similarly. The activity of the chosen window is set to true, and the activity of every other to false (**Listing 17**).

```
public void OpenSettings()
{
    _settingsView.SetActive(true);
    _mainMenuView.SetActive(false);
}
```

Listing 17: Main menu OpenSettings Method
Source: own elaboration

The only buttons with different functionalities are: the "Continue" and "Start game" buttons will open the level scene, and the "Quit" button will close the game (**Listing 19**). Entering the level will start a coroutine that will load the game level in the background (**Listing 18**), the player will then enter the level when the game is fully loaded. The "ContinueGame()" method works the same way as the "NewGame()" method, with the difference being that it does not call the Save system to clear the game save file.

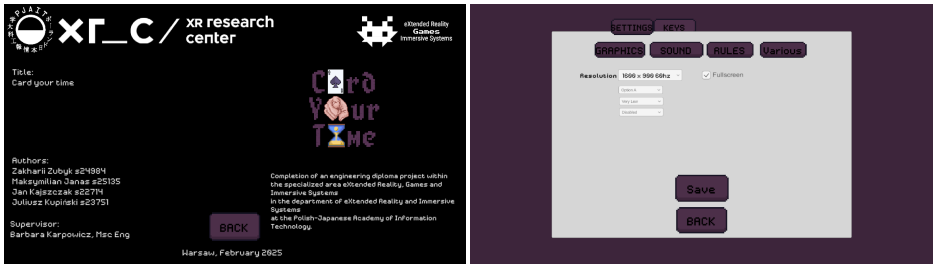
```
public IEnumerator NewGame()
{
    _levelLoading = true;
    SaveSystem.NewGame();
    var asyncLevelLoad = SceneManager.LoadSceneAsync(1);

    while (!asyncLevelLoad.isDone)
    {
        yield return null;
    }
}
```

Listing 18: Main menu NewGame Method
Source: own elaboration

```
public void QuitGame()
{
    if (_levelLoading) return;
    Application.Quit();
}
```

Listing 19: Main menu QuitGame Method
Source: own elaboration

Some of the other screens available to the players are: “Credits” (**see Fig. 48a**), “Compendium” (**see Fig. 48d**) and “Settings” (**see Fig. 48b and 48c**).



(a) Credits view of the Main menu
Source: own elaboration

(b) Visual settings view of the Main menu
Source: own elaboration

(c) Keys settings view of the Main menu
Source: own elaboration

(d) Compendium view of the Main menu
Source: own elaboration

Fig. 48: Screens of the Main menu
Source: own elaboration

Though not all have been implemented fully, these windows give the player basic access to some mechanics. The “Compendium” window in the future could hold information about cards or items gathered by the player, and enemies met by him. The “Settings” window gives the ability to change resolution or to switch between fullscreen and windowed play modes. And the last “Credits” window, displays among other things, some information about the team responsible for creating the game. All settings functionality has been implemented in the “SettingsManager.cs” class, and the “AppSettings.cs” class, the latter being a template for the json settings save file.

Changing the display modes between “fullscreen” and “windowed” is done by changing a bool variable inside the “Screen” class (**Listing 20**).

```
public void SetFullscreen(bool isFullscreen)
{
    Screen.fullScreen = isFullscreen;
}
```

Listing 20: Settings SetFullscreen Method
Source: How to Create a Settings Menu in Unity [1]

The resolution change mechanic is however much more complicated. First, a list of all available resolutions must be acquired. This is done with the use of the “Screen.resolutions” variable. Then all the resolutions are put into a drop-down menu inside the settings window. When a resolution is chosen by the player, a method is called (**Listing 21**) that will take an index of the chosen element inside the drop-down menu and set the resolution to the one corresponding to the given index inside the earlier acquired list of all possible resolutions.

```
public void SetResolution(int index)
{
    Resolution resolution = _resolutions[index];
    Screen.SetResolution(resolution.width, resolution.height, Screen.
    fullScreen);
    Application.targetFrameRate = Convert.ToInt32(math.round(resolution.
    refreshRateRatio.value));
}
```

Listing 21: Settings SetResolution Method
Source: How to Create a Settings Menu in Unity [1]

All settings are saved to a json file inside the “Application.StreamingAssetPath” directory and are loaded from it any time the game program is started by the user.

Now let us take a look at the “Pause” menu. It works similarly to the “Main” menu with the difference being, that it exists as a part of a game scene UI element, not as a separate scene. The

[1] https://www.red-gate.com/simple-talk/development/dotnet-development/how-to-create-a-settings-menu-in-unity/
[1] https://www.red-gate.com/simple-talk/development/dotnet-development/how-to-create-a-settings-menu-in-unity/

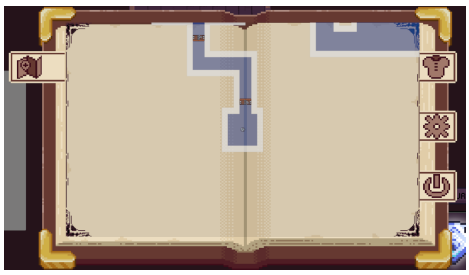
"IngameUIManager.cs" is the class mainly responsible for the menu operations, like closing and opening it.

The menu consists of four views. The main view is the "Pause" view (**see Fig. 49a**), opened and closed by pressing the "Esc" button. Thanks to this view the player is able to exit to the main menu or quit the game entirely, and also to pause the game when it is necessary. Another view is the "Inventory" view opened and closed with "E" key (**see Fig. 49b**), which has been already covered extensively in the "Inventory" subsection 5.1. Another view is the "Map" view opened and closed with "M" key (**see Fig. 49c**), showing a map of the dungeon the player is in currently. The last view is the "Game over" view (**see Fig. 49d**). This window can not be opened by the player, only losing the game can open it. When the window is open, the only way to exit it is to exit to main menu or quit the game entirely.

At the start of the game, all windows are deactivated. Each window is managed similarly, therefore to describe how it works we can use one of the windows as an example.

(a) Pause view of pause menu.
Source: own elaboration

(b) Inventory view of pause menu.
Source: own elaboration



(c) Map view of pause menu.
Source: own elaboration

(d) Game over view of pause menu.
Source: own elaboration

When the “E” button is pressed the inventory window opens. The “ChangeInventoryState()” (**Listing 22**) method is called. The method first checks if the animator of the menu windows is in use. If not, then it will check if the menu is not locked (the menu is locked when “Game over” screen is open). Next, if the inventory window is already open, the menu will close, otherwise, the menu will either change from a different view to the inventory view, or the whole menu will open on the inventory view.

```
private void ChangeInventoryState()
{
    if (_menuMenuAnimator.GetCurrentAnimatorStateInfo(0).IsTag("Book_close
    ") || _menuMenuAnimator.GetCurrentAnimatorStateInfo(0).IsTag("Book_open
    ")) return;
    if (_locked) return;
    if (!_inv.activeSelf)
    {
        Time.timeScale = 0;
        EventSystem.HideHand?.Invoke(true);
        OpenInventory();
        return;
    }
    CloseMenu();
}
```

Listing 22: ChangeInventoryState Method
Source: own elaboration

Opening of the inventory is done in the “OpenInventory()” (**Listing 23**) method. The function checks again if the menu is not locked, and then if it is active. If both are false, the function sets the right visibilities for all the menu windows and activates the right animator trigger.

```
public void OpenInventory()
{
    if (_locked) return;
    if (_inv.activeSelf) return;
    _minimap.SetActive(false);
    menuOpen = true;
    _menuView.SetActive(true);
    _mapView.SetActive(false);
    _pauseView.SetActive(false);
    _menuMenuAnimator.SetTrigger("Inventory");
}
```

Listing 23: OpenInventory Method
Source: own elaboration

After that, the animation will play. At the end of the animation, the animator will call a function called " ChangeInventoryVisible()" (**Listing 24**). The function will set the visibility of the elements of the chosen view, so that those elements are only visible from the moment the animation ends, not before.

```
public void ChangeInventoryVisible(bool visible = true)
{
    if (_locked) return;
    _inv.SetActive(visible);
    if (visible)
    {
        while (Inventory.Instance.notDisplayedYet.Count > 0)
        {
            var item = Inventory.Instance.notDisplayedYet[0];
            Inventory.Instance._itemsPanel.GetComponent<ListAllAvailable
    >().AddItemToList(item);
            Inventory.Instance.notDisplayedYet.Remove(item);
        }
    }
}
```

Listing 24: ChangeInventoryVisible Method
Source: own elaboration

At the end, the closing of the whole menu is done by the "Close-Menu()" function (**Listing 25**). The method will set the visibilities of all the views and save current equipment configuration. At the end, the correct animator trigger is called.

```
public void CloseMenu(){
    if (_locked) return;
    menuOpen = false;
    _inv.GetComponent<UIInventory>().SaveState();
    if (_inv.activeSelf)
        _inv.SetActive(false);

    _camera.m_Lens.OrthographicSize = _originalCameraSize;
    _pauseView.SetActive(false);
    _mapView.SetActive(false);
    Time.timeScale = 1;
    EventSystem.HideHand?.Invoke(false);
    _menuMenuAnimator.SetTrigger("Close");
}
```

Listing 25: CloseMenu Method
Source: own elaboration

The mechanics of all buttons in this menu are done the same way as in the Main menu. All are positioned on the sides of the window with all but the “Map” button being on the right side. The buttons will call the same methods as the corresponding key presses.

Lastly, let us take a closer look at the pause menu animator itself **(see Fig. 50)**. The animator consists of a couple of animations and a set of triggers. Each trigger starts one of the animations. At the end of each animation, a trigger **(see Fig. 51)** calls a function (**Listing 26**) that will call a corresponding function inside the menu manager script (**Listing 24**). That function will make the elements of the chosen view visible. It is done to ensure that the elements are not visible while a chosen window is not yet open.

```
1 public void SetInventoryVisible()
2 {
3     _IngameMenu.GetComponent<IngameUIManager>().ChangeInventoryVisible();
4 }
```

Listing 26: Example method setting menu window visible.
Source: own elaboration

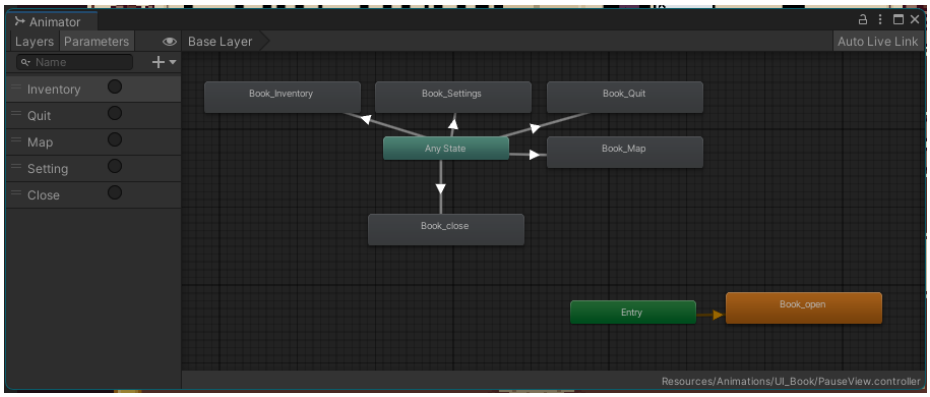


Fig. 50: Animator of the pause menu.
Source: own elaboration

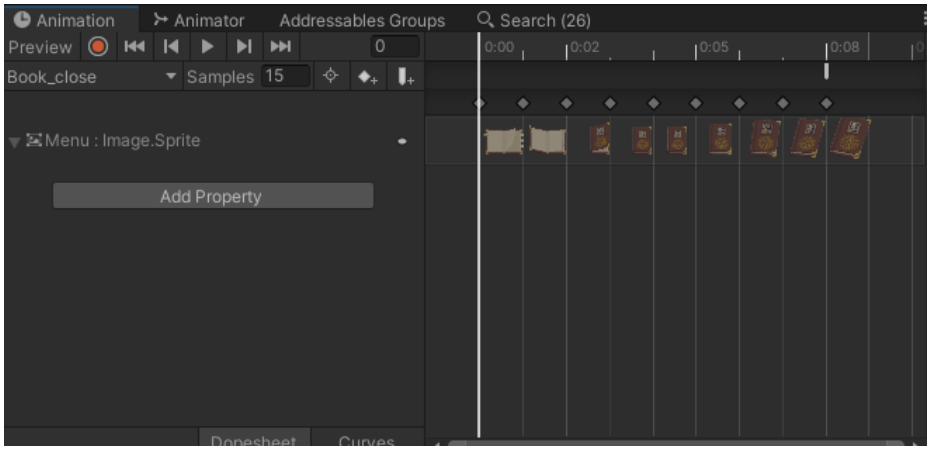


Fig. 51: Example animation of the pause menu.
Source: own elaboration

5.5 Input System Implementation

It was decided during project implementation that the best course of action would be to use the new "Input system" since it would unify the whole input reading and management structure.
All inputs are assigned a label and type, in our case either a normal button or Vector2D. Each input has a name and can have more than one button assigned (**see Fig. 52**). The names will be later used to create methods that the input system will call.

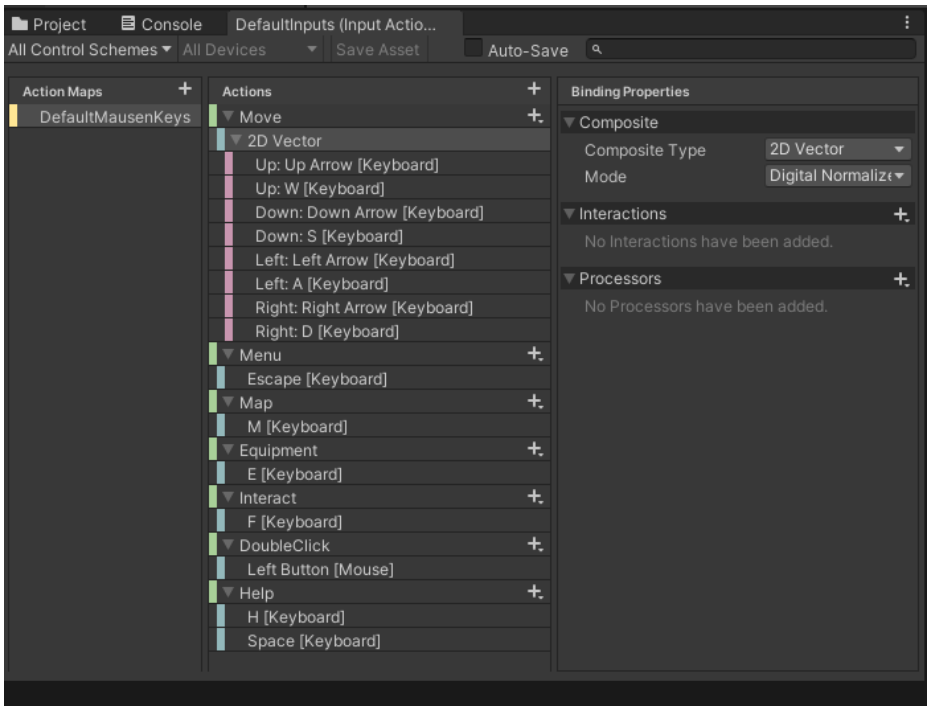


Fig. 52: The input system window
Source: own elaboration

After assigning the inputs, an interface is automatically created by the input system. "PlayerInputsController.cs" class has to implement that interface. The interface holds functions, which then must be implemented in our controller, and will be called once the right button is pressed.

The logic of the input system is held inside the "player" prefab in a class called "PlayerInputsController.cs". The whole class consists of many functions, all called by a key press, but the base logic of all of them is mostly the same, therefore only a couple of examples will be used.

The class must implement the interface created by the input controller, in the case of this project it is called "IDefaultMausenKeysActions". From this interface are taken the functions that will later be called by the input system.

Activation of the input system is done inside the "OnEnable()" function (**Listing 27**) called when the game object is activated. First, a "PrepareInputs()" method is called, which creates an instance of the "DefaultInputs" class and assigns it to the "controls" variable. Then inside the object, the interface is activated. Then each keybinding specified in the input system window has to be assigned a .performed function, and also if it is needed, a .canceled function.

In the "OnDisable()" function called when the object is deactivated, all of the methods are unassigned and the "controls" object is disabled, so it would not read any inputs when it should not do so.

```
1  private void OnEnable()
2  {
3      PrepareInputs();
4      _controls.DefaultMausenKeys.Move.performed += OnMove;
5      _controls.DefaultMausenKeys.Equipment.performed += OnEquipment;
6      _controls.DefaultMausenKeys.Equipment.canceled += OnEquipmentCanceled;
7      _controls.DefaultMausenKeys.Menu.performed += OnMenu;
8      _controls.DefaultMausenKeys.Menu.canceled += OnMenuCanceled;
9      _controls.DefaultMausenKeys.DoubleClick.performed += OnDoubleClick;
10     _controls.DefaultMausenKeys.Move.canceled += OnMoveCancelled;
11     _controls.DefaultMausenKeys.Help.performed += OnHelp;
12     _controls.DefaultMausenKeys.Map.performed += OnMap;
13     _controls.DefaultMausenKeys.Map.canceled += OnMapCanceled;
14 }
15
16 private void OnDisable()
17 {
18     _controls.Disable();
19     _controls.DefaultMausenKeys.Move.performed -= OnMove;
20     _controls.DefaultMausenKeys.Equipment.performed -= OnEquipment;
21     _controls.DefaultMausenKeys.Menu.performed -= OnMenu;
22     _controls.DefaultMausenKeys.DoubleClick.performed -= OnDoubleClick;
23     _controls.DefaultMausenKeys.Move.canceled -= OnMoveCancelled;
```

```
24      _controls.DefaultMausenKeys.Help.performed -= OnHelp;
25      _controls.DefaultMausenKeys.Map.performed -= OnMap;
26      _controls.DefaultMausenKeys.Equipment.canceled -= OnEquipmentCanceled;
27      _controls.DefaultMausenKeys.Menu.canceled -= OnMenuCanceled;
28      _controls.DefaultMausenKeys.Map.canceled -= OnMapCanceled;
29  }
30
31  private void PrepareInputs()
32  {
33      _controls = new DefaultInputs();
34      _controls.DefaultMausenKeys.Enable();
35  }
```

Listing 27: PlayerInputsController OnEnable OnDisable and PrepareInputs Methods
Source: own elaboration

As an example of an input function, the "OnMenu()" function (**Listing 28**) is called when the "Esc" button is pressed. The function will then give the "pressedButton" variable the true value, and call upon the "EventSystem" to open the pause menu window. The next method will give the bool variable the false value once the button is released. This logic is used only by the inputs connected to the in-game menu windows to block the player's ability to break logic by opening multiple windows at the same time. Other inputs consist of only one function and also use the "EventSystem" to call events and send values.

```
1  public void OnMenu(InputAction.CallbackContext context)
2  {
3      if (_buttonPressed) return;
4      _buttonPressed = true;
5      EventSystem.OpenClosePauseMenu?.Invoke();
6  }
7  public void OnMenuCanceled(InputAction.CallbackContext context)
8  {
9      _buttonPressed = false;
10 }
```

Listing 28: Menu inputs Methods
Source: own elaboration

A little bit of a different example is the "OnDoubleClick()" function (**Listing 29**). It makes use of raycasting to check if a pressed object is an item slot, and calls a method inside that item slot, differently than the rest of the function.

```
public void OnDoubleClick(InputAction.CallbackContext context)
{
    List<RaycastResult> results = new List<RaycastResult>();
    PointerEventData d = new PointerEventData(UnityEngine.EventSystems.
    EventSystem.current);
    d.position = Mouse.current.position.ReadValue();
    _raycaster.Raycast(d, results);
    foreach (RaycastResult r in results)
    {
        if(r.gameObject.CompareTag("ItemSlot"))
        {
            r.gameObject.GetComponent<ItemSlot>().DoubleClicked();
            break;
        }
    }
}
```

Listing 29: OnDoubleClick Method
Source: own elaboration

5.6 Addressables System Implementation

Addressables is an outside package that is used for asset management. Assets can be assigned as addressables, and put into groups (**see Fig. 53**). Each asset is also given tags, through which they will later be able to be accessed. All cards are given tags stating the type of the card, for example, “AttackCard”, and if a card is a default card a “DefaultCard” tag is also given. Items can be assigned an “Item” tag.

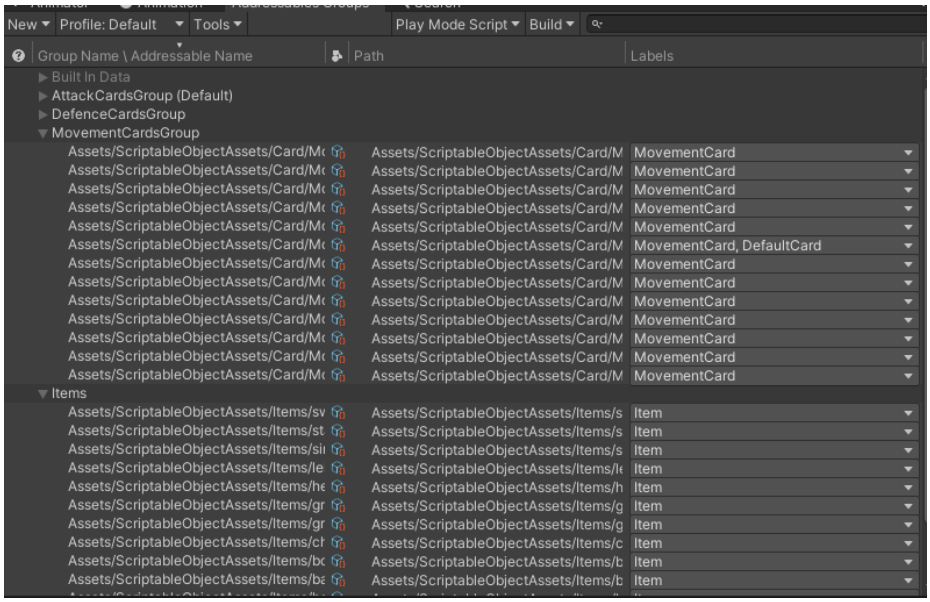


Fig. 53: Addressable assets list.
Source: own elaboration

AddressablesUtility is a script created to make assets easily accessible. A function called "LoadItems" (**Listing 30**) waits for addressables to return a list of objects that fit given tags. The function is accessible globally. The function requires a merge mode that specifies how the given tags will be treated, and an array of string tags. Most of the function have been provided by the "Addressables" system documentation.

```
1  public static List<object> LoadItems(Addressables.MergeMode mergeMode =
       Addressables.MergeMode.Union, params AddressablesTags[]
       addressablesTags)
2  {
3      if (addressablesTags.Length == 0) return null;
4
5      List<object> result = new List<object>();
6
7      List<string> _keys = new List<string>();
8
9      foreach (AddressablesTags tag in addressablesTags)
10     {
11         _keys.Add(tag.ToString());
12     }
```

```
13
14      AsyncOperationHandle<IList<object>> _loadHandle;
15
16      _loadHandle = Addressables.LoadAssetsAsync<object>(_keys,
17                                                         addressable =>
18          {
19              result.Add(addressable);
20          }, mergeMode, false);
21
22      _loadHandle.WaitForCompletion();
23      return result;
24  }
```

Listing 30: LoadItems Method
Source: Addressables documentation [1]

To keep things short, the function has two lists, one with tags given as an argument, and one for items to return. At the start of the method, all tags must be turned into strings. Then an async operation handler is created. This handler will hold a reference to an async method, which will return a list of objects labeled with given tags. When the handle has finished looking for objects, they are returned as a list.

Another function called "GetRandomItem" (**Listing 31**) returns a random item that has not yet been given to a player. The function works by first getting a list of all items by calling the "LoadItems" function. When the list of items is returned, all of them have to be converted to the "Item" object type. Then the function goes through all the items already in the player's possession and removes them from the list of all items. Then a random value is generated, being in the range of the "all items" lists current length. In the end, the item is added to the "Inventory.Instance.items" list, and to the "Inventory.Instance.notDisplayedYet" list if the equipment window is not open. If it is open, however, a function is called that displays that item in said window called "ListAllAvailable.AddItemToList".

[1] https://docs.unity3d.com/Packages/com.unity.addressables@2.3/manual/load-assets.html

```
public static void GetRandomItem()
{
    var items = LoadItems(Addressables.MergeMode.Union,AddressablesTags.
    Item );
    var inventory = Inventory.Instance;
    for (int i = 0; i < items.Count; i++)
    {
        var convItem = items[i] as Item;
        if (inventory.items.Contains(convItem))
        {
            items.Remove(items[i]);
            i--;
        }
    }
    if (items.Count <= 0) return;
    var index = UnityEngine.Random.Range(0,items.Count);
    if (index < 0)
    {
        return;
    }
    var result = items[index] as Item;
    Inventory.Instance.items.Add(result);
    if (!Inventory.Instance._itemsPanel.activeInHierarchy)
    {
        Inventory.Instance.notDisplayedYet.Add(result);
        return;
    }
    Inventory.Instance._itemsPanel.GetComponent<ListAllAvailable>().
    AddItemToList(result);
}
```

Listing 31: GetRandomItem Method
Source: own elaboration

The last method inside the "AddressablesUtilities" class is called "ItemsWithNames" (**Listing 32**), and is being used by the "SaveSystem" to give player items with names saved in a save file.

```
public static void ItemsWithNames(SaveTemplate save)
{
    Inventory.Instance.items.Clear();

    var items = LoadItems(Addressables.MergeMode.Union, AddressablesTags.
    Item);

    foreach (var item in items)
    {
        if(save.inventory.Contains((item as Item).itemName))
        {
            Inventory.Instance.items.Add(item as Item);
```

```
12          }
13          if (save.head != null && save.head == (item as Item).itemName)
14          {
15              Equipment.Instance.head = item as Item;
16          }
17          else if (save.chest != null && save.chest == (item as Item).
    itemName)
18          {
19              Equipment.Instance.chest = item as Item;
20          }
21          .
22          .
23          .
24          else if (save.item6 != null && save.item6 == (item as Item).
    itemName)
25          {
26              Equipment.Instance.item6 = item as Item;
27          }
28      }
29      if (!Inventory.Instance._itemsPanel.activeInHierarchy) return;
30      Inventory.Instance._itemsPanel.GetComponent<ListAllAvailable>().
    ListAllItemsInInv();
31  }
```

Listing 32: ItemsWithNames Method
Source: own elaboration

The way the function works is fairly simple. At first, the list of items inside the inventory is cleared, to be sure that all items will be added correctly. Then a list of all items is created via the "LoadItems" method. After this, a for loop goes over all the items and checks if their name is inside the "save" object given by the save system. If the object name exists in the given object items list, the item is added to the player's inventory. Then a place where the item was positioned is checked and rightly assigned. At the end, the system checks if the inventory window is displayed and if so, calls a function to display all the given items. The function is called "ListAllAvailable.ListAllItemsInInv()".

5.7 Save System

The save system is one of the most crucial systems in any game that has it. It lets the player continue his progress after closing the application. This game consists of a simple implementation of such system, the detailed description of which is explained below.

As to not overcomplicate our project too much, it has been decided that the save system will only give players a single save file, that is overwritten any time the player chooses to start a new game. The save system lets the player continue playing with already unlocked items, but it does not save any locations or variables other than a place where the item has been put inside the equipment.
The buttons in the main menu have been accustomed to the save system, making sure that the "continue" button does not appear on screen until the player first starts a game. At that point, a new save file is being created.

All of the system logic is contained in three classes, which also use some functions from others. The first class is called "SaveTemplate.cs" (**Listing 33**).

```
public class SaveTemplate
{
    public List<string> inventory = new List<string>();

    public string head;
    public string chest;
    public string legs;
    public string boots;
    public string rightHand;
    public string leftHand;

    public string item1;
    public string item2;
    public string item3;
    public string item4;
    public string item5;
    public string item6;
}
```

Listing 33: SaveTemplate Class
Source: own elaboration

The class consists of several string variables containing names of items that have been put in corresponding equipment slots, for example, the "head" variable holds the name of the item that has been equipped by a player in the head slot of the equipment. The class also holds a list of the names of all items currently unlocked by the player.

The purpose of the class is to be an object that can be easily serialized and written into a .json file, and then equally easily read from it.

The main logic of the save system is being held in the "SaveSystem.cs" class (**Listing 34**).

```
public static class SaveSystem
{
    private static string _saveGameFilePath = Application.
   persistentDataPath + "/Saves/save.sav";
```

Listing 34: SaveSystem Class
Source: own elaboration

The class is static, which gives free access to its methods from any place in the code, making them easier to use. At the beginning, a save file location is saved as a global variable.

Let us start by explaining how the game is saved. The "SaveGame()" (**Listing 35**) method is called any time the player enters a portal to another level. At the start, the method will check if a directory exists in the user's file system. The save file is stored in the user's "AppData" folder. If the folder exists, the method will then check if the save file itself also exists. If one or both do not exist they are immediately created.

```
public static void SaveGame(){
    FileStream file;
    if (!Directory.Exists(Application.persistentDataPath + "/Saves"))
       Directory.CreateDirectory(Application.persistentDataPath+"/Saves");
    if (File.Exists(_saveGameFilePath)) {
        file = File.OpenWrite(_saveGameFilePath);
    else
        file = File.Create(_saveGameFilePath);
```

Listing 35: Checking if save file exists.
Source: own elaboration

Next, a save object will be created and all of the names of the items will be added to it. They will also be put into variables corresponding to places in the equipment if any of the items are equipped (**Listing 36**).

```
SaveTemplate saveTemplate = new SaveTemplate();
file.Close();
foreach(var item in Inventory.Instance?.items)
{
    saveTemplate.inventory.Add(item.itemName);
}
var eq = Equipment.Instance;
if (eq.leftHand != null) saveTemplate.leftHand = eq.leftHand.itemName;
.
.
if (eq.item6 != null) saveTemplate.item6 = eq.item6.itemName;
```

Listing 36: Creating save object.
Source: own elaboration

At the end the save object is converted into json and written into the save file (**Listing 37**).

```
string save = JsonUtility.ToJson(saveTemplate,true);
System.IO.File.WriteAllText(_saveGameFilePath, save);
```

Listing 37: Writing to save file.
Source: own elaboration

The process of loading information from the save file is much simpler. It is being done by the "LoadGame()" method (**Listing 38**).

The method assumes that a save file has already been created by the user, since to be able to load the game the save file must exist in the right directory, otherwise, the "continue" button does not appear in the "main menu" scene.

The process of loading the game therefore consists of only a couple of instructions: reading the save file and deserializing it into a "SaveTemplate" object, which then can be sent to the "AddressablesUtilities" class, so that the right items can be there given to a player.

```
public static void LoadGame()
{
    var text = File.ReadAllText(_saveGameFilePath);
    var data = JsonUtility.FromJson<SaveTemplate>(text);
    AddressablesUtilities.ItemsWithNames(data);
}
```

Listing 38: LoadGame Method.
Source: own elaboration

Another method created for the system is the "NewGame()" method. The method's purpose is to create a new save file at the start of a new game, that does not hold any values. The method works mostly the same way as the "SaveGame()" method (**Listing 35**), only it does not read items from the player inventory, rather it saves an empty save file.

The last method used by the system is the "CheckIfSaveExists()" method. It is a function used mainly to check if a save file exists and display a "continue" button in the main menu in case it does.

5.8 Card Hand

Card hand is one of the most important parts of UI in every card game. It controls how the player views his playing hand and, depending on his currently drawn cards influences his in-game decisions. It consists of two main scripts that let the player establish the order of the cards and choose what card he wants to use in his current turn.

The first and foremost script is HandControllers.cs attached to the "hand" game object which is the parent object of every drawn card held in the hand. It allows the customization of how cards act and are set up in hand by the changing parameters in the inspector in Unity (**see Fig. 54**) to ensure that cards are visible and interactable.

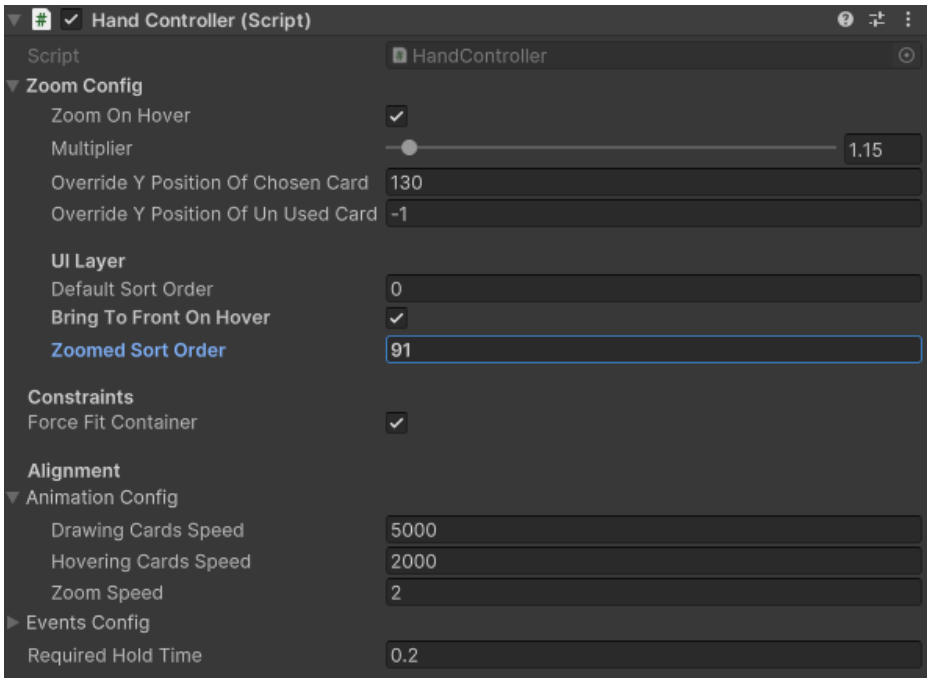


Fig. 54: HandController.cs inspector view
Source: own elaboration

To ensure that cards act as intended functions UpdateCards() is called every frame. It checks if any child objects weren't noticed in the previous frame (ex. newly drawn card). If yes, the InitCards() method is called which is responsible for setting up components of the card game object and adding it to the list of all cards in 'hand' by using the theSetUpCards() method and then setting its anchor in the SetCardsAnchor() method (**Listing 39**).
After setting up all possible new cards, the HandController calls the SetCardsPosition() method responsible for distributing all drawn cards at their calculated position that depends on the selected settings in the Unity inspector. Cards are evenly distributed either to fit the width of the 'hand' game object or to be placed just side by side without taking the width of the 'hand' into consideration. The next method: SetCardsUILayers() makes sure that the card on the right side will be always displayed over the card on the left side. It's important especially if the cards are fitted into their parent game object width and their total width is bigger. In this case, cards will overlap each other, and the correct layer setup will keep them readable. The last method UpdateCardOrder() controls the order of the cards from left to right. The player can rearrange the cards' order. The method makes sure that the card index in the main list of cards is adjusted and then on the next frame their position.

```
1  public class HandController : MonoBehaviour
2      {
3          private RectTransform _rectTransform;
4          private List<Wrapper> _cards = new();
5
6          [SerializeField] private ZoomConfig _zoomConfig;
7
8          [Header("Constraints")] [SerializeField]
9          private bool _forceFitContainer;
10
11         public static readonly int attackCardLimit = 2;
12         public static int defenceCardLimit = 2;
13         public static int movementCardLimit = 1;
14
15         public static int cardLimit = attackCardLimit + defenceCardLimit +
       movementCardLimit;
16
17         public static int currentCardNumber;
18         public static int currentAttackCardNumber;
```

```
19          public static int currentDefenceCardNumber;
20          public static int currentMovementCardNumber;
21
22          [Header("Alignment")] [SerializeField] private AnimationConfig
    _animationConfig;
23
24          private Wrapper _currDraggedCard;
25
26          [SerializeField] private EventsConfig _eventsConfig;
27
28
29          public float requiredHoldTime = 1.5f;
30
31          private void Start()
32          {
33              _rectTransform = GetComponent<RectTransform>();
34              InitCards();
35              EventSystem.DestroyCard.AddListener(DestroyCard);
36              EventSystem.HideHand.AddListener(DisableHand);
37              EventSystem.RemoveHand.AddListener(RemoveHand);
38          }
39
40
41          private void InitCards()
42          {
43              SetUpCards();
44              SetCardsAnchor();
45          }
46
47          private void Update()
48          {
49              UpdateCards();
50          }
51
52          private void SetUpCards()
53          {
54              _cards.Clear();
55              foreach (Transform card in transform)
56              {
57                  var wrapper = card.GetComponent<Wrapper>();
58                  if (wrapper == null)
59                  {
60                      wrapper = card.gameObject.AddComponent<Wrapper>();
61                  }
62
63                  _cards.Add(wrapper);
64
65                  AddOtherComponentsIfNeeded(wrapper);
66
```

```
67                  wrapper.zoomConfig = _zoomConfig;
68                  wrapper.handController = this;
69                  wrapper.eventsConfig = _eventsConfig;
70                  wrapper.animationConfig = _animationConfig;
71              }
72          }
73
74          private void AddOtherComponentsIfNeeded(Wrapper wrapper)
75          {
76              var canvas = wrapper.GetComponent<Canvas>();
77              if (canvas == null)
78              {
79                  canvas = wrapper.gameObject.AddComponent<Canvas>();
80              }
81
82              canvas.overrideSorting = true;
83              if (wrapper.GetComponent<GraphicRaycaster>() == null)
84              {
85                  wrapper.gameObject.AddComponent<GraphicRaycaster>();
86              }
87          }
88
89          private void SetCardsAnchor()
90          {
91              foreach (Wrapper card in _cards)
92              {
93                 card.SetAnchor(new Vector2(0, 0.5f), new Vector2(0, 0.5f));
94              }
95          }
96
97          private void UpdateCards()
98          {
99              if (transform.childCount != _cards.Count)
100             {
101                 InitCards();
102             }
103
104             if (_cards.Count == 0)
105             {
106                 return;
107             }
108
109             currentCardNumber = currentAttackCardNumber  +
    currentDefenceCardNumber + currentMovementCardNumber;
110
111             SetCardsPosition();
112             SetCardsUILayers();
113             UpdateCardOrder();
114         }
```

```
115
116         private void UpdateCardOrder()
117         {
118             if (_currDraggedCard == null)
119             {
120                 return;
121             }
122
123             var draggedCardIndex = _cards.Count(card => _currDraggedCard.
    transform.position.x > card.transform.position.x);
124             var originalCardIndex = _cards.IndexOf(_currDraggedCard);
125             _currDraggedCard.transform.SetSiblingIndex(draggedCardIndex);
126             if (draggedCardIndex != originalCardIndex)
127             {
128                 _cards.RemoveAt(originalCardIndex);
129                 if (draggedCardIndex > originalCardIndex &&
    draggedCardIndex < _cards.Count - 1)
130                 {
131                     draggedCardIndex--;
132                 }
133             }
134
135             _currDraggedCard.transform.SetSiblingIndex(draggedCardIndex);
136         }
137
138         private void SetCardsUILayers()
139         {
140             for (int i = 0; i < _cards.Count; i++)
141             {
142                 _cards[i].uiLayer = _zoomConfig.defaultSortOrder + i;
143             }
144         }
145
146         private void SetCardsPosition()
147         {
148             var cardsTotalWidth = _cards.Sum(card => card.Width * card.
    transform.lossyScale.x);
149             var containerWidth = _rectTransform.rect.width * transform.
    lossyScale.x;
150             if (_forceFitContainer && cardsTotalWidth > containerWidth)
151             {
152                 DistributeChildrenToFitContainer(cardsTotalWidth);
153             }
154             else
155             {
156                 DistributeChildrenWithoutOverlap(cardsTotalWidth);
157             }
158         }
159
```

```
160         private void DistributeChildrenToFitContainer(float
    cardsTotalWidth)
161         {
162             var width = _rectTransform.rect.width * transform.lossyScale.x;
163             var distanceBetweenCards = (width - cardsTotalWidth) / (_cards
    .Count - 1);
164             var currX = transform.position.x - width / 2;
165             foreach (Wrapper card in _cards)
166             {
167                 var correctCardWidth = card.Width * card.transform.
    lossyScale.x;
168                 card.targetPosition = new Vector2(currX + correctCardWidth
     / 2, transform.position.y);
169                 currX += correctCardWidth + distanceBetweenCards;
170             }
171         }
172 
173         private void DistributeChildrenWithoutOverlap(float
    cardsTotalWidth)
174         {
175             var currPosition = transform.position.x - cardsTotalWidth / 2;
176             foreach (Wrapper card in _cards)
177             {
178                 var adjustedChildWidth = card.Width * card.transform.
    localScale.x;
179                 card.targetPosition = new Vector2(currPosition +
    adjustedChildWidth / 2, transform.position.y);
180                 currPosition += adjustedChildWidth;
181             }
182         }
183     }
```

Listing 39: HandController card Set up
Source: own elaboration

The second script controlling the card hand system is Wrapper.cs (**Listing 40**), which is attached to every drawn card by HandController.cs. It's responsible for all interactions between the player's mouse cursor and cards in the 'hand'. It includes events like: hover, press, and drag. On hover, i.e. placing the cursor over a card's game object in the game the selected card overrides its Y position to stand out in the hand and changes its layer, so it's displayed over any cards that may cover it. When the cursor leaves the card's border it goes back to its previous position. By pressing the left mouse button on the card, it changes its 'state' to selected, overrides its Y position, sends its range data read from the card to the range finder algorithm,

and allows the player to confirm the action by choosing the correct target. To distinguish pressing action from hovering action, apart from overriding its Y position it also makes the rest of the cards in the hand less visible. If the player presses the right mouse button the selection will undo and all cards will go back to their previous state. Only one card can selected. On drag, i.e. pressing and holding cards with the left mouse button for a time set up in HandCotrollers' inspector card will stick to the mouse cursor until the mouse button is released. This action allows the player to re-arrange the order of cards in hand.

```
public void OnPointerEnter(PointerEventData eventData)
{
    if (cardInUse == this)
    {
        return;
    }
    if (isDragged)
    {
        return;
    }

    if (zoomConfig.bringToFrontOnHover)
    {
        _canvas.sortingOrder = zoomConfig.zoomedSortOrder;
    }

    eventsConfig?.cardHover?.Invoke(new CardHover(this));
    isHovered = true;
}

public void OnPointerExit(PointerEventData eventData)
{
    if (cardInUse == this)
    {
        return;
    }
    if (isDragged)
    {
        return;
    }

    _canvas.sortingOrder = uiLayer;
    isHovered = false;
    eventsConfig.cardUnHover.Invoke(new CardUnhover(this));
}
```

```
36
37 public void OnPointerDown(PointerEventData eventData)
38 {
39     if (Input.GetMouseButtonDown(0))
40     {
41         _isHeld = true;
42         if(isDragged)
43         {
44             _dragStartPos = new Vector2(transform.position.x - eventData.
    position.x,
45                 transform.position.y - eventData.position.y);
46             eventsConfig?.cardUnHover?.Invoke(new CardUnhover(this));
47             return;
48         }
49
50         handController.OnCardDragStart(this);
51
52         if (PlaceHolder.isTaken)
53         {
54             cardInUse.isHovered = false;
55             eventsConfig?.cardUnHover?.Invoke(new CardUnhover(this));
56             EventSystem.HideRange.Invoke();
57         }
58         eventsConfig?.cardHover?.Invoke(new CardHover(this));
59         cardInUse = this;
60         PlaceHolder.isTaken = true;
61         int range = Convert.ToInt32(GetComponent<CardDisplay>().range.text
    );
62         EventSystem.ShowRange.Invoke(range);
63     }
64
65     if (Input.GetMouseButtonDown(1))
66     {
67         if (cardInUse == this)
68         {
69             DeselectCard();
70         }
71     }
72 }
73
74 private void DeselectCard()
75 {
76         eventsConfig?.cardUnHover?.Invoke(new CardUnhover(this));
77         isHovered = false;
78         cardInUse = null;
79         PlaceHolder.isTaken = false;
80         EventSystem.HideRange.Invoke();
81 }
82
```

```
public void OnPointerUp(PointerEventData eventData)
{
    isDragged = false;
    _isHeld = false;
    _timeHeld = 0f;
    handController.OnCardDragEnd();
}
```

Listing 40: Wrapper.cs action methods
Source: own elaboration

5.9 Visual and Playable Deck

For card hand meaning, there has to be a deck from which cards can be drawn. The deck system takes a list of cards which was generated by the inventory, creates usable card game objects, and draws them into the 'hand' when the player's turn starts.
DeckController.cs (**Listing 41**) receives a list of cards every time the player makes a change in his equipment outside of the combat. When the player enters the room CreateDeck() method is called and card objects are instantiated. DeckController also holds a limitation for the number of all and each type of card that can be in the card hand. To ensure that cards aren't exceeding the given limit, the deck inside is made out of 3 lists of cards, each storing cards of a different type i.e. attack, defense, and movement. When the DrawACard() method is called cards of the type that are missing from hands are randomly sent to the card hand and are deleted from the lists. If the list runs out of cards, cards get replenished so the player doesn't have to worry about running out of them.

```
public class DeckController : MonoBehaviour
{
    private void DownloadCardsFromInventory()
    {
        _cards = _equipment.cards;
    }

    public void ManageDeck()
    {
        DownloadCardsFromInventory();
        Debug.Log(CombatMode.isPlayerInCombat);
        if (_attackDeck.Count != 0 && !CombatMode.GetIsPlayerInCombat())
        {
           UpdateDeck();
```

```
15          }
16          if (_defenceDeck.Count != 0 && !CombatMode.GetIsPlayerInCombat())
17          {
18              UpdateDeck();
19          }
20          if (_movementDeck.Count != 0 && !CombatMode.GetIsPlayerInCombat())
21          {
22              UpdateDeck();
23          }
24          if (!CombatMode.GetIsPlayerInCombat())
25          {
26              CreateDeck();
27          }
28      }
29
30      private void UpdateDeck()
31      {
32          RemoveCardsFromDeck();
33          DownloadCardsFromInventory();
34          CreateDeck();
35      }
36      private void RemoveCardsFromDeck()
37      {
38          foreach (Transform child in transform)
39          {
40              Destroy(child.gameObject);
41          }
42
43          _movementDeck.Clear();
44          _attackDeck.Clear();
45          _defenceDeck.Clear();
46      }
47
48      private void CreateDeck()
49      {
50          foreach (var card in _cards)
51          {
52             var newCard = Instantiate(_cardPrefab, transform, true);
53             newCard.GetComponent<CardDisplay>().cardSO = card;
54             var position = transform.position;
55             newCard.transform.position = new Vector2(position.x - _overlay,
    position.y);
56             newCard.transform.localScale = new Vector3(1, 1, 1);
57             switch (card.type)
58             {
59                 case CardType.Attack: _attackDeck.Add(newCard);
60                     break;
61                 case CardType.Defense: _defenceDeck.Add(newCard);
62                     break;
```

```
63                case CardType.Movement: _movementDeck.Add(newCard);
64                    break;
65            }
66        }
67
68        _deckExists = true;
69        Destroy(_createDeckText);
70    }
71
72    public void DrawACard()
73    {
74        do
75        {
76            if (HandController.currentAttackCardNumber < attackCardLimit)
77            {
78                SendCardToHand(CardType.Attack);
79            }
80            if (HandController.currentDefenceCardNumber < defenceCardLimit
    )
81            {
82                SendCardToHand(CardType.Defense);
83            }
84            if (HandController.currentMovementCardNumber <
    movementCardLimit)
85            {
86                SendCardToHand(CardType.Movement);
87            }
88
89            HandController.currentCardNumber++;
90        }while (HandController.currentCardNumber < cardLimit);
91    }
92
93    private void SendCardToHand(CardType type)
94    {
95        if (_attackDeck.Count == 0 || _defenceDeck.Count == 0 ||
    _movementDeck.Count == 0)
96        {
97            UpdateDeck();
98        }
99
100       GameObject card;
101       switch (type)
102       {
103           case CardType.Attack:
104               card = _attackDeck[Random.Range(0, _attackDeck.Count)];
105               _attackDeck.Remove(card);
106               card.transform.SetParent(_hand.transform);
107               HandController.currentAttackCardNumber++;
108               break;
```

```
109             case CardType.Defense:
110                 card = _defenceDeck[Random.Range(0, _defenceDeck.Count)];
111                 _defenceDeck.Remove(card);
112                 card.transform.SetParent(_hand.transform);
113                 HandController.currentDefenceCardNumber++;
114                 break;
115             case CardType.Movement:
116                 card = _movementDeck[Random.Range(0, _movementDeck.Count)
    ];
117                 _movementDeck.Remove(card);
118                 card.transform.SetParent(_hand.transform);
119                 HandController.currentMovementCardNumber++;
120                 break;
121         }
122     }
123 }
```

Listing 41: HandController card Set up
Source: own elaboration

5.10 Applying Card Effects

The system lets the player make use of his current equipment by making the cards it generates usable on enemies and himself, depending on which cards he draws into his hand and what his strategy is.
The system is controlled by ApplyCardsEffect.cs (**Listing 42**) for the attack and defense types of cards due to their influence on the health points of game characters. The script is connected to the 'body' part of each in-game character to make use of its collider component. If the player is in combat mode. the card is selected, and the target is within the range of the card, using the left mouse button on the unit will the activate ApplyCardsEffect.cs script, applying the health points change to the character, deleting the used card and showing the health change over the target's head.

```
1 public class ApplyCardEffect : MonoBehaviour
2 {
3     private void OnMouseDown()
4     {
5         if (!Input.GetMouseButtonDown(0) || !PlayerController.
    getPlayerTurn()) return;
6         if (isOnRangedTile()) return;
7         if (!PlaceHolder.isTaken) return;
8
```

```
        var cardInfo = Wrapper.GetCardCurrentCardInfo();

        SendHpModification(cardInfo);
        _timersManager.PlayedAttackCard(_healthBarScript.gameObject,
    _hpSliderGameObj);
    }
    private void SendHpModification(Wrapper cardInfo)
    {
        var card = cardInfo.display.cardSO;
        var cost = card.cost;
        var hpChange = card.damage;
        hpChange = (card.type == CardType.Attack ? hpChange * -1 :
    hpChange);
        _healthBarScript.ChangeHealth(hpChange);
        _timersManager.ChangeActiveTimerValue(cost);
        AnimateSourceEntity(card.type);
        ShowPopUpDamage(hpChange);
        EventSystem.DestroyCard.Invoke();
        EventSystem.LogAction?.Invoke(card);
    }
    private void ShowPopUpDamage(int hpChange)
    {
        var popUpPosition = new Vector2(transform.position.x, transform.
    position.y + 1);
        GameObject popUp = Instantiate(popUpPrefab, popUpPosition,
    Quaternion.identity);
        var text = popUp.GetComponentInChildren<TMP_Text>();
        string popUpText;
        if (hpChange > 0)
        {
            popUpText = "+" + hpChange;
            text.color = Color.green;
            ShowHitAnimation();
        }
        else
        {
            popUpText = hpChange.ToString();
            text.color = Color.red;
            ShowHitAnimation();
        }
        text.text = popUpText;
    }
    private bool isOnRangedTile()
    {
        return !MouseController._rangeTiles.Contains(standingOnTile);
    }
}
```

Listing 42: HandController card Set up
Source: own elaboration

The rest of the system, which applies movement card effects, can be found in MouseController.cs (**Listing 43**) and PlayerController.cs scripts. Movement cards only affect the player's movement and don't influence any health bar of any entity.

```
public class MouseController : MonoBehaviour
{
    void LateUpdate()
    {
        if (Input.GetMouseButtonDown(0))
        {
      overlayTile.ShowTile();
            if (CombatMode.isPlayerInCombat && Wrapper.cardInUse != null)
            {
                if (Wrapper.cardInUse.display.cardSO.type == CardType.
    Movement)
                {
                    _playerController.PlayerCardMovement(overlayTile,
    _rangeTiles);
                }
            }
        }

        if (Input.GetMouseButtonDown(1))
        {
            if (_playerController != null && !CombatMode.isPlayerInCombat)
            {
                _playerController.PlayerMouseMovement(overlayTile);
            }
            overlayTile.HideTile();
            }
        }
    }
}
```

Listing 43: MouseController.cs
Source: own elaboration

5.11 Health Bar Changes Log

The main focus of the system was to improve the information flow from the game to the player (**see Fig. 55**), so he could run back his actions in case of miss-click or miscalculations.

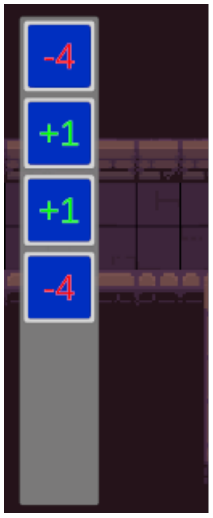


Fig. 55: Actions Log
Source: own elaboration

Whenever there is an event using a card, the HistoryController.cs (**Listing 44**) script is called, to instantiate the history tile containing information about the previous card used and about the type of health modification it did.

```
public class HistoryController : MonoBehaviour
{
    private void CreateLog(CardsSO cardInfo)
    {
        var newTile = Instantiate(historyTile, transform, true);
        newTile.transform.localScale = new Vector3(1, 1, 1);
        var tileText = newTile.GetComponentInChildren<TMP_Text>();

        if (cardInfo.type == CardType.Attack)
        {
            tileText.color = Color.red;
            tileText.text = "-" + cardInfo.damage;
```

```
13          }
14          else if (cardInfo.type == CardType.Defense)
15          {
16              tileText.color = Color.green;
17              tileText.text = "+" + cardInfo.damage;
18          }
19          else
20          {
21              tileText.color = Color.white;
22              tileText.text = cardInfo.damage.ToString();
23          }
24
25          _tilesCounter++;
26          UpdateSize();
27      }
28  }
```

Listing 44: Actions Log
Source: own elaboration

5.12 Ground-Mouse Interaction

The goal of creating an interaction between the mouse and the map (**see Fig. 56**) where the game's action is happening was to make the player feel, that he has control over what's happening on the screen.

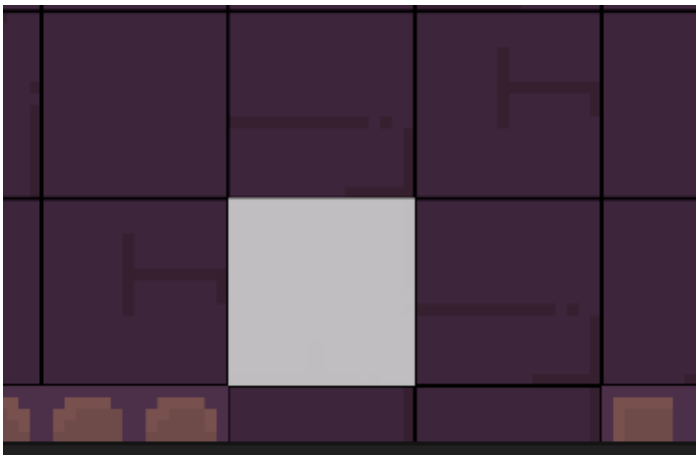


Fig. 56: Highlighted tile
Source: own elaboration

Every tile that belongs to the ground layer of the tilemap has an overlay. Each overlay has the same width and length as the ground tile, so it acts like it's a single tile itself. All overlays are generated by OverlayManager.cs (**Listing 45**) after all the rooms and their tilemaps have been generated. DungeonBuilder.cs calls the CreateOverlaysForRooms() method, passing two lists: one of all ground tilemaps and one of all wall tilemaps created. First, the algorithm uses wall tilemaps to filter any possible ground tilemaps that might have been generated in the same position as a wall. In the end, it iterates through all ground tiles it receives, generating an overlay of game objects at the world position of each tile.

```
1  public class OverlayManager : MonoBehaviour
2  {
3      public void CreateOverlaysForRooms(List<Tilemap> groundTilemaps, List<
       Tilemap> wallTilemaps)
4      {
5          if (overlayContainer.transform.childCount != 0)
6          {
7              ResetOverContainer();
8          }
9
10         map = new Dictionary<Vector2, OverlayTile>();
11
12         foreach (var groundTm in groundTilemaps)
13         {
14             var wallTm = wallTilemaps[groundTilemaps.IndexOf(groundTm)];
15             BoundsInt bounds = groundTm.cellBounds;
16
17             for (int z = bounds.max.z; z > bounds.min.z; z--)
18             {
19                 for (int y = bounds.min.y; y < bounds.max.y; y++)
20                 {
21                     for (int x = bounds.min.x; x < bounds.max.x; x++)
22                     {
23                         var groundTileLocation = new Vector3Int(x, y, z);
24                         var cellWorldPosition = groundTm.
       GetCellCenterWorld(groundTileLocation);
25                         if (wallTm.HasTile(wallTm.WorldToCell(
       cellWorldPosition))) //check if there is a wall tile on the same spot
       so the overlay tile doesn't generate in walls
26                         {
27                             continue;
28                         }
29
30                         var tileKey = new Vector2((float)Math.Round(
       cellWorldPosition.x,3), (float)Math.Round(cellWorldPosition.y,3));
```

```
                        if (groundTm.HasTile(groundTileLocation - new
    Vector3Int(0, 0, 1)) && groundTm.gameObject
                                .GetComponent<TilemapRenderer>()
            .sortingLayerName.Equals("Ground"))
                        {
                            if (!map.ContainsKey(tileKey))
                            {
                                var overlayTile = Instantiate(
    overlayPrefab, overlayContainer.transform);
                                overlayTile.transform.position = new
    Vector3(cellWorldPosition.x, cellWorldPosition.y, cellWorldPosition.z +
     1);
                                overlayTile.gameObject.GetComponent<
    OverlayTile>().gridLocation = tileKey;
                                map.Add(tileKey, overlayTile.gameObject.
    GetComponent<OverlayTile>());
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Listing 45: Overlays generation
Source: own elaboration

To create interaction with the overlays every frame raycast is cast on the position of the mouse's cursor. If the raycast hits an overlay tile it changes its color. If the left mouse button is pressed the color stays until a different tile gets selected (**Listing 46**).

```
public class MouseController : MonoBehaviour
{
    private void LateUpdate()
    {
        RaycastHit2D? hit = GetFocusedOnTile();

        if (hit.HasValue)
        {
            OverlayTile overlayTile = hit.Value.collider.gameObject.
    GetComponent<OverlayTile>();

            if (overlayTile == null)
            {
                return;
            }
```

```
15
16          if (overlayTile.GetComponent<SpriteRenderer>() != null)
17          {
18              cursor.transform.position = overlayTile.transform.position;
19              cursor.GetComponent<SpriteRenderer>().sortingOrder =
20                overlayTile.GetComponent<SpriteRenderer>().sortingOrder;
21
22
23               if (Input.GetMouseButtonDown(0))
24               {
25                   overlayTile.ShowTile();
26
27                   if (CombatMode.isPlayerInCombat && Wrapper.cardInUse
    != null)
28                   {
29                       if (Wrapper.cardInUse.display.cardSO.type ==
    CardType.Movement)
30                       {
31                           _playerController.PlayerCardMovement(
    overlayTile, _rangeTiles);
32                       }
33                   }
34               }
35
36               if (Input.GetMouseButtonDown(1))
37               {
38                   if (_playerController != null && !CombatMode.
    isPlayerInCombat)
39                   {
40                      _playerController.PlayerMouseMovement(overlayTile);
41                   }
42                   overlayTile.HideTile();
43               }
44          }
45      }
46      if (_playerController.standingOnTile == null)
47      {
48          _playerController.standingOnTile = _playerController.
    GetCurrentTile();
49      }
50  }
51  private RaycastHit2D? GetFocusedOnTile()
52  {
53      Vector3 mousePos = Camera.main.ScreenToWorldPoint(Input.
    mousePosition);
54      Vector2 mousePos2d = new Vector2(mousePos.x, mousePos.y);
55
56      RaycastHit2D[] hits = Physics2D.RaycastAll(mousePos2d, Vector2.
    zero);
```

```
57
58          if (hits.Length > 0)
59          {
60              return hits.OrderByDescending(i => i.collider.transform.
    position.z).First();
61          }
62
63          return null;
64      }
65  }
```

Listing 46: Mouse position rayscast cast
Source: own elaboration

5.13 Range Finding

Limiting the range (**see Fig. 57**) of the player's cards is key to keeping the combat part of the game fluid and challenging. Without it, the player would be able to defeat enemies with no need to approach them and that would make the game too easy and as a result, it would become boring after a short amount of time.

Fig. 57: Highlighted range tiles of card with range = 2
Source: own elaboration

The range-finding algorithm is divided into two scripts. The first script called RangeFinder.cs (**Listing 48**) is responsible for creating a list of overlay tiles that should be displayed as a range of the card. It loops as many times as the card's range value. Starting from the overlay tile where the player stands (**Listing 47**), the algorithm goes through all the tiles that were added to the rangeTiles list in the previous iteration, expanding the range of the action until it reaches its targeted range.

```
if (Input.GetMouseButtonDown(0))
{
    overlayTile.ShowTile();
    if (CombatMode.isPlayerInCombat && Wrapper.cardInUse != null)
    {
        if (Wrapper.cardInUse.display.cardSO.type == CardType.Movement)
        {
            _playerController.PlayerCardMovement(overlayTile, _rangeTiles);
        }
    }
}

if (Input.GetMouseButtonDown(1))
{
    if (_playerController != null && !CombatMode.isPlayerInCombat)
    {
        _playerController.PlayerMouseMovement(overlayTile);
    }
    overlayTile.HideTile();}
```

Listing 47: Fragment of MouseController.cs LateUpdate() method where selected tile is sent to PlayerController.cs
Source: own elaboration

The second script that takes part in range-finding is OverlayManager.cs (**Listing 49**). It has the GetRangeTiles() method that uses the dictionary of all the overlay tiles, which is located in the OverlayManager.cs. The method returns the list of tiles that are around the tile Rangefinder.cs is currently iterating through. It returns four tiles depending on their position to the position of the overlay tile passed as a parameter to the method: position.x+1, position.x-1, position.y+1, position.y-1.

```
public class RangeFinder
{
    public List<OverlayTile> GetTilesInRange(int range, OverlayTile playersTile)
    {
        var rangeTiles = new List<OverlayTile>();

        int counter = 0;
        var tilesFromPreviousCount = new List<OverlayTile> {playersTile};

        rangeTiles.Add(playersTile);

        while (counter < range)
        {
            var surroundingTiles = new List<OverlayTile>();

            foreach (var tile in tilesFromPreviousCount)
            {
                surroundingTiles.AddRange(OverlayManager.Instance.GetRangeTiles(new Vector2(tile.gridLocation.x, tile.gridLocation.y)));
            }

            rangeTiles.AddRange(surroundingTiles);
            tilesFromPreviousCount = surroundingTiles.Distinct().ToList();
            counter++;
        }


        return rangeTiles.Distinct().ToList();
    }
}
```

Listing 48: RangeFinder.cs
Source: own elaboration

```
public class OverlayManager : MonoBehaviour
{
    public List<OverlayTile> GetRangeTiles(Vector2 playerOccupiedTile)
    {
        var rangeTiles = new List<OverlayTile>();

        Vector2 playerPos = new Vector2((float)Math.Round(playerOccupiedTile.x, 3), (float)Math.Round(playerOccupiedTile.y, 3));

        rangeTiles.Add(map[playerPos]);

        Vector2 tileToCheck = new Vector2((float)Math.Round(playerOccupiedTile.x + 1,3), (float)Math.Round(playerOccupiedTile.y,3));
```

```
12          if (map.ContainsKey(tileToCheck))
13          {
14              rangeTiles.Add(map[tileToCheck]);
15          }
16
17          tileToCheck = new Vector2((float)Math.Round(playerOccupiedTile.x
    ,3), (float)Math.Round(playerOccupiedTile.y + 1,3));
18
19          if (map.ContainsKey(tileToCheck))
20          {
21              rangeTiles.Add(map[tileToCheck]);
22          }
23
24          tileToCheck = new Vector2((float)Math.Round(playerOccupiedTile.x -
     1,3), (float)Math.Round(playerOccupiedTile.y,3));
25
26          if (map.ContainsKey(tileToCheck))
27          {
28              rangeTiles.Add(map[tileToCheck]);
29          }
30
31          tileToCheck = new Vector2((float)Math.Round(playerOccupiedTile.x
    ,3), (float)Math.Round(playerOccupiedTile.y - 1,3));
32
33          if (map.ContainsKey(tileToCheck))
34          {
35              rangeTiles.Add(map[tileToCheck]);
36          }
37
38          return rangeTiles;
39      }
40  }
```

Listing 49: Fragment of OverlayManager.cs returning rangetiles to the RangeFinder.cs
Source: own elaboration

5.14 Mouse-Controlled Movement

Throughout most project iterations, the player's movement was controlled by either arrow keys or WASD keys. In the end, it was decided that the movement should be controlled by the right mouse click. The player clicks on the place where he wants his character to go. Depending on whether the player is in combat or not, the MouseController.cs script receives input from the mouse click and sends it directly to the PlayerController.cs (**Listing 50**), where it is directed

to the PathFinder.cs script, which contains two path finding algorithms: one for outside-of-combat movements and one for in-combat movement.

PathFinder.cs (**Listing 51**) is directly used inside of the PlayerController.cs script. It returns a list called currentPath, containing overlays that belong to the path found by PathFinder.cs algorithm. If the path exists which results in the currentPath list not being empty the LateUpdate() method (called by the engine at the end of every frame) calls the MoveAlongPath() method. In MoveAlongPath() position of the first overlay is assigned as the player's character new position and is deleted from the list. Process repeats itself until currentPath list becomes empty.

```
1  public class PlayerController : MonoBehaviour
2  {
3      public void PlayerMouseMovement(OverlayTile overlayTile)
4      {
5          _currentPath = _pathFinder.FindPathOutsideOfCombat(standingOnTile,
       overlayTile);
6      }
7
8      public void PlayerCardMovement(OverlayTile overlayTile, List<
       OverlayTile> rangeTiles)
9      {
10         if (_currentPath.Count > 0)
11         {
12             return;
13         }
14
15         if (CombatMode.isPlayerInCombat)
16         {
17             _currentPath = _pathFinder.FindPathInCombat(standingOnTile,
       overlayTile, rangeTiles);
18             if (_currentPath.Count == 0)
19             {
20                 return;
21             }
22             EventSystem.DestroyCard?.Invoke();
23         }
24     }
25
26     private void MoveAlongPath()
27     {
28         if (Wrapper.cardInUse != null)
```

```
29          {
30              return;
31          }
32          if (_currentPath[0] == null)
33          {
34              return;
35          }
36          transform.position = Vector2.MoveTowards(transform.position,
    _currentPath[0].transform.position, _moveSpeed * Time.deltaTime);
37
38          if (Vector2.Distance(transform.position, _currentPath[0].transform
    .position) < 0.0001f)
39          {
40              PlaceOnTile(_currentPath[0]);
41              _currentPath.RemoveAt(0);
42          }
43      }
44
45      private void PlaceOnTile(OverlayTile overlayTile)
46      {
47          var position = overlayTile.transform.position;
48          var closestTile = transform.position = new Vector3(position.x,
    position.y + 0.0001f, position.z);
49              standingOnTile = overlayTile;
50      }
51  }
```

Listing 50: PlayerController.cs responsible for repositioning the player's character
Source: own elaboration

The two path-finding algorithms used in PathFinder.cs calculate the path in the same way. The difference between them is the overlay tiles they are allowed to use to create the path. When the player is inside the combat, PathFinder is only allowed to use overlay tiles that are in the range of the player's movement card, which was calculated by the range-finding algorithm in RangeFinder.cs. The outside-of-combat path-finding doesn't have that restriction, it can use all the overlay tiles it thinks are needed.

```
public class PathFinder : MonoBehaviour
{
    private Dictionary<Vector2, OverlayTile> searchableTiles;

    public List<OverlayTile> FindPathOutsideOfCombat(OverlayTile start, OverlayTile end)
    {
        List<OverlayTile> possibleNodes = new List<OverlayTile>();
        List<OverlayTile> chosenNodes = new List<OverlayTile>();

        possibleNodes.Add(start);

        while (possibleNodes.Count > 0)
        {
            OverlayTile currentOverlayTile = possibleNodes.OrderBy(x => x.F).First();

            possibleNodes.Remove(currentOverlayTile);
            chosenNodes.Add(currentOverlayTile);

            if (currentOverlayTile == end)
            {
                return GetFinishedList(start, end);
            }

            foreach (var tile in GetNeighbourOverlayTiles(currentOverlayTile))
            {
                if (chosenNodes.Contains(tile) || Mathf.Abs(currentOverlayTile.transform.position.z - tile.transform.position.z) > 1)
                {
                    continue;
                }

                tile.G = GetManhattenDistance(start, tile); // G = Distance from starting tile
                tile.H = GetManhattenDistance(end, tile);  // H = Distance from target tile

                tile.Previous = currentOverlayTile;

                if (!possibleNodes.Contains(tile))
                {
                    possibleNodes.Add(tile);
                }
            }
        }

```

```
43          return new List<OverlayTile>();
44      }
45
46      public List<OverlayTile> FindPathInCombat(OverlayTile start,
    OverlayTile end, List<OverlayTile> inRangeTiles)
47      {
48          if (!inRangeTiles.Contains(end))
49          {
50              return new List<OverlayTile>();
51          }
52          searchableTiles = new Dictionary<Vector2, OverlayTile>();
53          List<OverlayTile> possibleNodes = new List<OverlayTile>();
54          HashSet<OverlayTile> chosenNodes = new HashSet<OverlayTile>();
55
56          if (inRangeTiles.Count > 0)
57          {
58              foreach (var item in inRangeTiles)
59              {
60                  searchableTiles.Add(item.grid2DLocation, OverlayManager.
    Instance.map[item.grid2DLocation]);
61              }
62          }
63
64          possibleNodes.Add(start);
65          while (possibleNodes.Count > 0)
66          {
67              OverlayTile currentOverlayTile = possibleNodes.OrderBy(x => x.
    F).First();
68
69              possibleNodes.Remove(currentOverlayTile);
70              chosenNodes.Add(currentOverlayTile);
71
72              if (currentOverlayTile == end)
73              {
74                  return GetFinishedList(start, end);
75              }
76
77              foreach (var tile in GetNeighbourOverlayTiles(
    currentOverlayTile))
78              {
79                  if (chosenNodes.Contains(tile) || Mathf.Abs(
    currentOverlayTile.transform.position.z - tile.transform.position.z) >
    1)
80                  {
81                      continue;
82                  }
83
84                  tile.G = GetManhattenDistance(start, tile); // G =
    Distance from starting tile
```

```
85                  tile.H = GetManhattenDistance(end, tile); // H = Distance
    from target tile
86                  tile.Previous = currentOverlayTile;
87                  if (!possibleNodes.Contains(tile))
88                  {
89                      possibleNodes.Add(tile);
90                  }
91              }
92          }
93          return new List<OverlayTile>();
94      }
95
96      private List<OverlayTile> GetFinishedList(OverlayTile start,
    OverlayTile end)
97      {
98          List<OverlayTile> finishedList = new List<OverlayTile>();
99          OverlayTile currentTile = end;
100
101         while (currentTile != start)
102         {
103             finishedList.Add(currentTile);
104             currentTile = currentTile.Previous;
105         }
106
107         finishedList.Reverse();
108
109         return finishedList;
110     }
111
112     private float GetManhattenDistance(OverlayTile start, OverlayTile tile
    )
113     {
114         return Mathf.Abs(start.gridLocation.x - tile.gridLocation.x) +
    Mathf.Abs(start.gridLocation.y - tile.gridLocation.y);
115     }
116
117     private List<OverlayTile> GetNeighbourOverlayTiles(OverlayTile
    currentOverlayTile)
118     {
119         var map = OverlayManager.Instance.map;
120
121         List<OverlayTile> neighbours = new List<OverlayTile>();
122
123         Vector2 tileToCheck = new Vector2((float)Math.Round(
    currentOverlayTile.gridLocation.x + 1,3), (float)Math.Round(
    currentOverlayTile.gridLocation.y,3));
124
125         if (map.ContainsKey(tileToCheck))
126         {
```

```
127             neighbours.Add(map[tileToCheck]);
128         }
129 
130         tileToCheck = new Vector2((float)Math.Round(currentOverlayTile.
    gridLocation.x - 1,3), (float)Math.Round(currentOverlayTile.
    gridLocation.y,3));
131 
132         if (map.ContainsKey(tileToCheck))
133         {
134             neighbours.Add(map[tileToCheck]);
135         }
136 
137         tileToCheck = new Vector2((float)Math.Round(currentOverlayTile.
    gridLocation.x ,3), (float)Math.Round(currentOverlayTile.gridLocation.y
     + 1,3));
138 
139         if (map.ContainsKey(tileToCheck))
140         {
141             neighbours.Add(map[tileToCheck]);
142         }
143 
144         tileToCheck = new Vector2((float)Math.Round(currentOverlayTile.
    gridLocation.x ,3), (float)Math.Round(currentOverlayTile.gridLocation.y
     - 1,3));
145 
146         if (map.ContainsKey(tileToCheck))
147         {
148             neighbours.Add(map[tileToCheck]);
149         }
150 
151         return neighbours.Distinct().ToList();
152     }
153 }
```

Listing 51: PathFinder.cs algorithms
Source: own elaboration

The algorithm itself is a greedy best-first search algorithm, that starts from the overlay tile the player is standing on and checks four surrounding overlay tiles: position.x+1, position.x-1, position.y+1, position.y-1, then chooses the overlay tile closest to the target overlay tile and moves on to it. Calculation repeats itself until the target overlay tile is reached. As a result, a list of overlay tiles is created which is later assigned to the currentPath list in PlayerController.cs.

5.15 Camera System

In order for the game to be able to properly track the players actions, we needed to implement a camera system, along with the ability to adjust certain features and functions to correct its operation to our vision. For this purpose, we used the Cinemachine package of the Unity engine in combination with a script that describes the properties of the camera and its functions.

The CameraProperties.cs script is located inside the "Camera" game object, while the actual Camera device and CinemachineBrain component for providing parameters from an active Virtual Camera to the main Camera device are placed inside the child game object of the "Camera" game object, called "Main Camera". The CinemachineVirtualCamera component, which represents the Virtual Camera, is placed in another child object of the "Camera" object, called "Virtual Camera".

5.15.1 Camera Zoom

Camera zoom properties are described inside the CameraProperties.cs script **Listing 52**. Here, we declare and initialize variables containing references to Virtual Camera and players Transform, as well as the variables holding values for the upper and lower zoom limits, the speed of zoom and the value of the previous zoom. Inside the ZoomCamera() method, we assign the value of the current size of the camera (which projection is set to Orthographic in the Inspector window) in the "currentZoom" variable, which is then used inside the Clamp() method that returns the new value of the camera size between the minimal and maximum zoom limits. Then we overwrite the current value of the camera size with the new value. The ZoomCamera() method is called in the Update() method, where it takes the value of the virtual axis coming from the mouse wheel. Due to the fact that Update() is called with each subsequent frame, it is possible to simulate a camera zoom this way.

```
public class CameraProperties : MonoBehaviour
{
    public CinemachineVirtualCamera virtualCamera;
    public Transform player;
    public float zoomSpeed = 2f;
    public float minZoom = 1f;
    public float maxZoom = 100f;
    private float _previousZoom;

    /*
     *
     */
    private void Update()
    {
        float zoomInput = Input.GetAxis("Mouse ScrollWheel");
        ZoomCamera(zoomInput);
    }

    private void ZoomCamera(float increment)
    {
        float currentZoom = virtualCamera.m_Lens.OrthographicSize;
        float newZoom = Mathf.Clamp(currentZoom - increment * zoomSpeed,
    minZoom, maxZoom);
        virtualCamera.m_Lens.OrthographicSize = newZoom;
    }
    /*
     *
     */
}
```

Listing 52: Fragment of CameraProperties.cs script with Update() and ZoomCamera() methods
Source: own elaboration

5.15.2 Camera Close-ups of the Rooms at the Beginning of Combat

Whenever the player gets inside the room and combat starts, the camera performs a close-up on the centre of the room, setting up the view on the whole room as an introduction for the player to familiarize with the layout of the enemy units. This is achieved by the AdjustCameraToTheRoomSize() method **Listing 53**, where we assign a new value to the camera size and adjust the camera position based on the size of the collider bounding area inside the room and the position of its centre.

```
public class CameraProperties : MonoBehaviour
{
    public CinemachineVirtualCamera virtualCamera;
    public Transform player;
    public float zoomSpeed = 2f;
    public float minZoom = 1f;
    public float maxZoom = 100f;
    private float _previousZoom;

    /*
     *
     */

    public void AdjustCameraToTheRoomSize(Vector2 size, Vector3 centre)
    {
        virtualCamera.m_Lens.OrthographicSize = MathF.Max(size.x, size.y);
        virtualCamera.transform.position = new Vector3(centre.x, centre.y,
    centre.z);
    }
}
```

Listing 53: Fragment of CameraProperties.cs script with AdjustCameraToTheRoomSize() method
Source: own elaboration

This method is called inside the OnTriggerEnter2D() method of the InstantiatedRoom class **Listing 54**. If the player finds himself in a combat situation, the reference to the room Composite Collider is obtained through the GetComponentInChildren() method, as well as the position of the room centre. The variables that hold this data are provided as arguments to the AdjustCameraToTheRoomSize() method. Then, we instantiate an empty object in the centre of the room through InstantiateRoomCenterObject() and set the Virtual Camera to follow that object, basically moving the camera to the centre of the room.

```
public class InstantiatedRoom : MonoBehaviour
{

    /*
     *
     */

    private CameraProperties _camera;

    private void Awake()
    {
```

```
12          /*
13           *
14           */
15          _camera = FindObjectOfType<CameraProperties>();
16      }
17
18      /*
19       *
20       */
21
22      private void OnTriggerEnter2D(Collider2D collision)
23      {
24          if (enemyInRoomList.Count == 0 && !collision.gameObject.CompareTag
    ("Player")) return;
25          EventSystem.StartCountdown.Invoke();
26          CloseAllDoors();
27          EventSystem.StopPath.Invoke();
28          if (CombatMode.isPlayerInCombat)
29          {
30              CompositeCollider2D compCollider = GetComponentInChildren<
    CompositeCollider2D>();
31              Vector3 roomWorldCenterCords = compCollider.bounds.center;
32              //Debug.LogError(this.collision.transform.parent.parent.name);
33              _camera.AdjustCameraToTheRoomSize(new Vector2(
34                      compCollider.bounds.size.x / 2,
35                      compCollider.bounds.size.y / 2),
36                  roomWorldCenterCords);
37              GameObject roomCenterObject = InstantiateRoomCenterObject(
    roomWorldCenterCords);
38              _camera.virtualCamera.Follow = roomCenterObject.transform;
39          }
40      }
41
42      /*
43       *
44       */
45
46      private GameObject InstantiateRoomCenterObject(Vector3 roomCenter)
47      {
48          GameObject roomCenterObject = new GameObject("RoomCenterObject");
49          roomCenterObject.transform.position = roomCenter;
50          roomCenterObject.transform.SetParent(collision.transform);
51          return roomCenterObject;
52      }
53  }
```

Listing 54: Fragment of InstantiatedRoom.cs script with OnTriggerEnter2D() method

Source: own elaboration

Once the combat is finished and the player leaves the room, we set the Virtual Camera to follow the player character inside the OnTriggerExit2D() method **Listing 55**.

```
private void OnTriggerExit2D(Collider2D collision)
    {
        if (!CombatMode.isPlayerInCombat)
        {
            _camera.virtualCamera.Follow = _camera.player;
        }
    }
```

Listing 55: OnTriggerExit2D() method of InstantiatedRoom class
Source: own elaboration

5.16 Pathfinding for Enemy Units Based on A* Algorithm

During the development of the video game, we encountered a question on how to make the enemy units move in correlation to the changing environment during the turn-based combat scenarios, such as when the player decides to move their character to the other side of the room, or when an event occurs, which adds other obstacles on the map - to make them adapt to the different situations during gameplay. We have concluded that we could apply a pathfinding algorithm as part of the enemy AI system inside the game - this would provide the needed adaptability in the enemy's behavior.

Since the field of pathfinding algorithms consists of its various types and their variants - specifically designed for different uses and with their list of advantages and disadvantages - we needed to analyze individual solutions and choose the one that suits our needs and the structure of the in-game world before proceeding with our implementation. While researching pathfinding algorithms, we focused on finding the best possible compromise between overall performance and the simplicity of the rules behind the algorithm. Ultimately, we chose the A* algorithm, which we used in the game.

The A* algorithm is a search algorithm with wide applications in different problems related to graph traversals and pathfinding. It uses a heuristic approach to find the shortest path from the starting position to the destined one. In our approach, we used the Manhattan distance **Listing 56** between two nodes **Listing 57** as a

heuristic. We also used references to two tilemaps inside the rooms where enemies are placed **Listing 58**. This way, we could specify where the nodes should be generated, so they are not placed on unwalkable structures, such as walls. These references are found and implemented in the EnemyController.cs script **Listing 59**, which is also one of the components inside the enemy prefab, designed to describe enemy behavior and actions.

```
private float GetDistance(Node nodeA, Node nodeB)
    {
        var distanceX = Mathf.Abs(nodeA.gridPosition.x - nodeB.
    gridPosition.x);
        var distanceY = Mathf.Abs(nodeA.gridPosition.y - nodeB.
    gridPosition.y);

        return distanceX + distanceY;
    }
```

Listing 56: GetDistance() method
Source: own elaboration

```
private class Node
    {
        public Vector3 worldPosition;
        public Vector3Int gridPosition;
        public Node parent;
        public float gCost;
        public float hCost;

        public float FCost => gCost + hCost;

        public Node(Vector3 worldPosition, Vector3Int gridPosition)
        {
            this.worldPosition = worldPosition;
            this.gridPosition = gridPosition;
        }
    }
```

Listing 57: Node class describing the node object
Source: own elaboration

```
public class Pathfinding : MonoBehaviour
{
    [HideInInspector] public Tilemap groundTilemap;
    [HideInInspector] public Tilemap topTilemap;

    private Dictionary<Vector3Int, Node> _nodes = new Dictionary<
    Vector3Int, Node>();
```

```
7       private HashSet<Vector3Int> _busyTilesSet = new HashSet<Vector3Int>();
8
9       /*
10      ...
11      */
12  {
```

Listing 58: Declaration of two Tilemap variables
Source: own elaboration

```
1   public class EnemyController : MonoBehaviour
2   {
3
4       /*
5       ...
6       */
7
8       private Pathfinding _pathfinding;
9
10      /*
11      ...
12      */
13
14      void Start()
15      {
16          /*
17          ...
18          */
19
20          _pathfinding = GetComponent<Pathfinding>();
21          if (_pathfinding == null)
22          {
23              Debug.LogError("Pathfinding script not found on the Enemy
        object");
24          }
25
26          /*
27          ...
28          */
29
30          Transform gridTransform = transform.parent;
31          Transform walkableTilemapTransform = gridTransform.Find("Ground");
32          Transform collisionTilemapTransform = gridTransform.Find("
        CollisionTOP");
33
34          _pathfinding.groundTilemap = walkableTilemapTransform.GetComponent
        <Tilemap>();
35          _pathfinding.topTilemap = collisionTilemapTransform.GetComponent<
        Tilemap>();
```

```
36
37          if (walkableTilemapTransform != null)
38          {
39              Debug.Log("Walkable tilemap accessed successfully.");
40          }
41          else
42          {
43              Debug.LogError("Failed to access walkable tilemap.");
44
45          }
46
47          if (collisionTilemapTransform != null)
48          {
49              Debug.Log("Collision tilemap accessed successfully.");
50          }
51          else
52          {
53              Debug.LogError("Failed to access collision tilemap.");
54
55          }
56
57          /*
58          ...
59          */
60      }
61
62      /*
63      ...
64      */
65
66  }
```

Listing 59: Implementation of Tilemap variables inside the Start() method in EnemyController.cs script
Source: own elaboration

In the FindPath() method **Listing 60**, we perform all of the pathfinding calculations, using methods for creating and placing nodes based on information provided by Tilemap variables and enemies' positions in the room where they are placed **Listing 61**, searching nodes' neighbors **Listing 62**, returning the retraced path to the target **Listing 63**, and Manhattan distance calculations mentioned earlier.

```
public List<Vector3> FindPath(Vector3 targetPosition)
{
    if (groundTilemap == null || topTilemap == null)
    {
        // Debug.LogWarning("Ground or top tilemap not set.");
        return null;
    }


    CreateNodesForEnemies();


    var startCell = groundTilemap.WorldToCell(transform.position);
    var targetCell = groundTilemap.WorldToCell(targetPosition);

    Vector3 thisObjectWorldPosition = transform.position;

    _nodes.Add(
        groundTilemap.WorldToCell(thisObjectWorldPosition),
        new Node(thisObjectWorldPosition, groundTilemap.WorldToCell(
    thisObjectWorldPosition))
        );

    if (!_nodes.ContainsKey(startCell) || !_nodes.ContainsKey(targetCell))
    {
        // Debug.LogWarning("Start or target not selected");
        return null;
    }

    var openNodeList = new List<Node>();
    var closedNodeSet = new HashSet<Node>();
    openNodeList.Add(_nodes[startCell]);

    while (openNodeList.Count > 0)
    {
        var currentNode = openNodeList[0];
        for (int i = 1; i < openNodeList.Count; i++)
        {
            if (openNodeList[i].FCost < currentNode.FCost ||
                openNodeList[i].FCost == currentNode.FCost &&
                openNodeList[i].hCost < currentNode.hCost)
            {
                currentNode = openNodeList[i];
            }
        }

        openNodeList.Remove(currentNode);
        closedNodeSet.Add(currentNode);

```

```
49          if (currentNode == _nodes[targetCell])
50          {
51              return RetracePath(_nodes[startCell], _nodes[targetCell]);
52          }
53
54          foreach (var neighbor in GetNeighbors(currentNode))
55          {
56              if (closedNodeSet.Contains(neighbor) || neighbor == null)
57              {
58                  continue;
59              }
60
61              var newMovementCostToNeighbor = currentNode.gCost +
    GetDistance(currentNode, neighbor);
62              if (newMovementCostToNeighbor < neighbor.gCost || !
    openNodeList.Contains(neighbor))
63              {
64                  neighbor.gCost = newMovementCostToNeighbor;
65                  neighbor.hCost = GetDistance(neighbor, _nodes[targetCell])
    ;
66                  neighbor.parent = currentNode;
67
68                  if (!openNodeList.Contains(neighbor))
69                  {
70                      openNodeList.Add(neighbor);
71                  }
72              }
73          }
74      }
75
76      return null;
77  }
```

Listing 60: FindPath() method
Source: own elaboration

```
1  private void CreateNodesForEnemies()
2  {
3      _nodes.Clear();
4      _busyTilesSet.Clear();
5
6      EnemyController[] enemies = FindObjectsOfType<EnemyController>();
7
8      foreach (EnemyController enemy in enemies)
9      {
10         Vector3 enemyPosition = enemy.transform.position;
11         Vector3Int tilePosition = groundTilemap.WorldToCell(enemyPosition);
12         _busyTilesSet.Add(tilePosition);
13     }
```

```
14
15      foreach (var pos in groundTilemap.cellBounds.allPositionsWithin)
16      {
17          var worldPos = groundTilemap.GetCellCenterWorld(pos);
18          bool isTileBusy = _busyTilesSet.Contains(pos);
19          if (groundTilemap.HasTile(pos) && !topTilemap.HasTile(pos) && !
    isTileBusy)
20          {
21              _nodes[pos] = new Node(worldPos, pos);
22          }
23      }
24      // Debug.Log($"Created {_nodes.Count} nodes");
25  }
```

Listing 61: CreateNodesForEnemies() method
Source: own elaboration

```
1   private List<Node> GetNeighbors(Node node)
2   {
3       var neighbors = new List<Node>();
4       var directions = new Vector3Int[]
5       {
6           Vector3Int.up,
7           Vector3Int.down,
8           Vector3Int.left,
9           Vector3Int.right
10      };
11
12      foreach (var dir in directions)
13      {
14          var neighborPosition = node.gridPosition + dir;
15          if (_nodes.ContainsKey(neighborPosition) && _nodes[
    neighborPosition] != null)
16          {
17              neighbors.Add(_nodes[neighborPosition]);
18          }
19      }
20
21      return neighbors;
22  }
```

Listing 62: GetNeighbors() method
Source: own elaboration

```
private List<Vector3> RetracePath(Node startNode, Node endNode)
{
    var path = new List<Vector3>();
    var currentNode = endNode;
    while (currentNode != startNode)
    {
        path.Add(currentNode.worldPosition);
        currentNode = currentNode.parent;
    }
    path.Reverse();
    return path;
}
```

Listing 63: RetracePath() method
Source: own elaboration

5.17 Random Dungeon Generation Algorithm

The random dungeon generation algorithm is crafted to deliver dynamic and engaging layouts for 2D game (see Fig. 59). Its primary aim is to improve replayability by ensuring each playthrough presents a unique level pattern. Utilizing hierarchical graphs, randomization, and pre-designed templates (see Fig. 58), the system achieves varied designs while maintaining logical coherence.

The Room Node Graph system organizes dungeon layouts through a hierarchical structure of interconnected nodes. Each node represents a room, while edges denote the corridors linking them. This system comprises the following key components:

- **RoomNodeGraphSO**: Oversees the entire graph structure **Listing 65**.
- **RoomNodeSO**: Represents an individual room within the graph. After what it's groups into levels **Listing 67**.
- **RoomTemplateSO**: Specifies the visual design and functional characteristics of each room **Listing 66**.

To create a map layout each potential dungeon layout was created. Then, after a level, random graph was selected. Each node within the graph has assigned a template of dungeon's room with all visual and gameplay elements. After establishing the room node graph, the graph gets traversed **Listing 64** through and on each node assigned part of the room is instantiated.

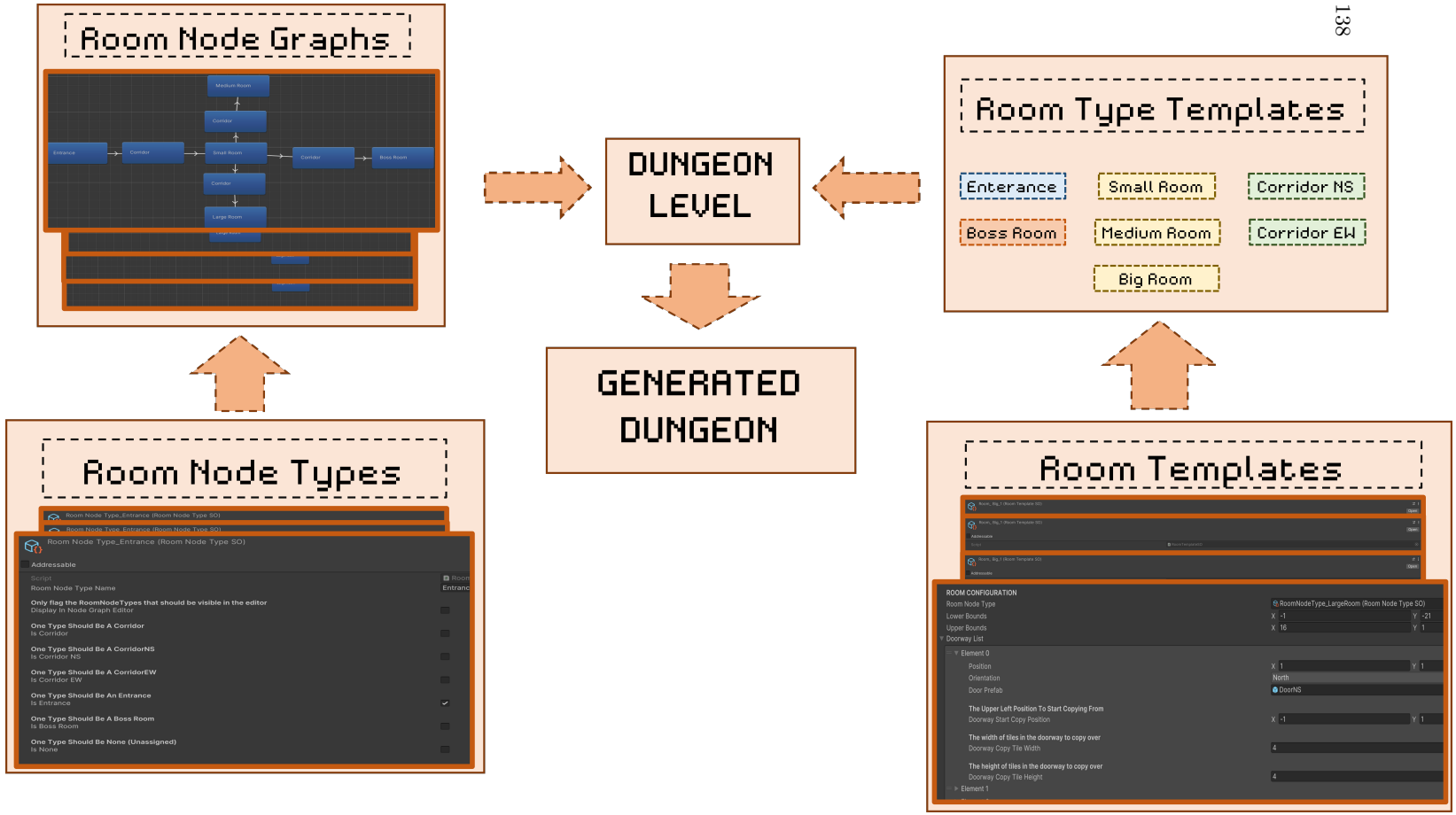


Fig. 58: How Dungeon Builder Work
Source: own elaboration

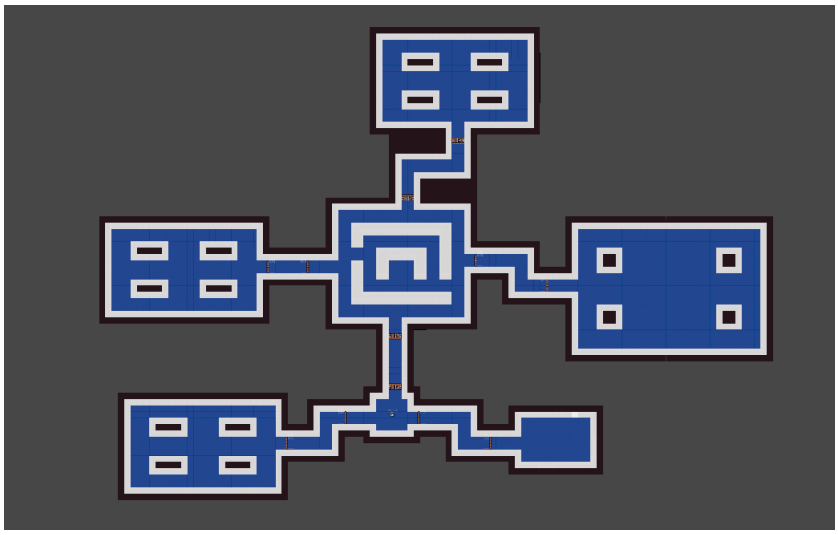


Fig. 59: Final Result of Generated Dungeon
Source: own elaboration

```
void GenerateDungeon(Graph dungeonGraph) {
    foreach (var node in dungeonGraph.Nodes) {
        RoomTemplate roomTemplate = AssignTemplate(node.RoomType);
        PlaceRoom(node.Position, roomTemplate);
    }
}

RoomTemplate AssignTemplate(RoomType type) {
    // Logic to select the correct template based on room type
    return templateDatabase.GetTemplate(type);
}

void PlaceRoom(Vector2 position, RoomTemplate template) {
    // Instantiate room based on template and place it at the specified
    position
    Instantiate(template.Prefab, position, Quaternion.identity);
}
```

Listing 64: Traversing a Graph to Generate Dungeon Rooms
Source: own elaboration

```
[CreateAssetMenu(fileName = "RoomNodeGraph", menuName = "Scriptable
    Objects/Dungeon/Room Node Graph")]
public class RoomNodeGraphSO : ScriptableObject
{
```

```
    [HideInInspector] public RoomNodeTypeListSO roomNodeTypeList;
    [HideInInspector] public List<RoomNodeSO> roomNodeList = new List<RoomNodeSO>();
    [HideInInspector] public Dictionary<string, RoomNodeSO> roomNodeDictionary = new Dictionary<string, RoomNodeSO>();

    private void Awake()
    {
        LoadRoomNodeDictionary();
    }

    // Load the room node dictionary
    private void LoadRoomNodeDictionary()
    {
        roomNodeDictionary.Clear();
        foreach (RoomNodeSO node in roomNodeList)
        {
            roomNodeDictionary[node.ID] = node;
        }
    }

    // Get room node by type
    public RoomNodeSO GetRoomNode(RoomNodeTypeSO roomNodeType)
    {
        foreach (RoomNodeSO node in roomNodeList)
        {
            if (node.roomNodeType == roomNodeType) return node;
        }
        return null;
    }

    // Get room node by ID
    public RoomNodeSO GetRoomNode(string roomNodeID)
    {
        roomNodeDictionary.TryGetValue(roomNodeID, out RoomNodeSO roomNode);
        return roomNode;
    }
}
```

Listing 65: Core functionality of RoomNodeGraphSO
Source: own elaboration

```
[CreateAssetMenu(fileName = "Room_", menuName ="Scriptable Objects/Dungeon/Room")]
public class RoomTemplateSO : ScriptableObject
{
   [HideInInspector] public string guid;

```

```
6     [Header("ROOM PREFAB")]
7     [Tooltip("The prefab for the room containing all tilemaps and
      environment objects")]
8     public GameObject prefab;
9
10    [HideInInspector] public GameObject previousPrefab; // Regenerate guid
      if prefab changes
11
12    [Header("ROOM CONFIGURATION")]
13    [Tooltip("The room node type, corresponding to the room node graph.
      Includes exceptions for corridors (NS and EW).")]
14    public RoomNodeTypeSO RoomNodeType;
15
16    [SerializeField] public List<Doorway> DoorwayList;
17 }
```

Listing 66: Core functionality of RoomTemplateSO
Source: own elaboration

```
1  public class DungeonLevelSO : ScriptableObject
2  {
3     [Header("BASIC LEVEL DETAILS")]
4     [Tooltip("The name for the level")]
5     public string levelName;
6
7     [Header("ROOM TEMPLATES FOR LEVEL")]
8     [Tooltip("List of room templates to include in the level. Ensure all
      room node types in the graphs are covered.")]
9     public List<RoomTemplateSO> roomTemplateList;
10
11    [Header("ROOM NODE GRAPHS FOR LEVEL")]
12    [Tooltip("List of room node graphs to be randomly selected for the
      level.")]
13    public List<RoomNodeGraphSO> roomNodeGraphList;
14
15 #if UNITY_EDITOR
16    // Validate scriptable object details
17    private void OnValidate()
18    {
19       HelperUtilities.ValidateCheckEmptyString(this, nameof(levelName),
      levelName);
20       HelperUtilities.ValidateCheckEnumerableValues(this, nameof(
      roomTemplateList), roomTemplateList);
21       HelperUtilities.ValidateCheckEnumerableValues(this, nameof(
      roomNodeGraphList), roomNodeGraphList);
22
23       // Validation checks for corridors and entrance types
24       bool isEWCorridor = false, isNSCorridor = false, isEnterance = false
      ;
```

```
25
26          // Loop through room templates and node graphs for validation
27          foreach (RoomNodeGraphSO graph in roomNodeGraphList)
28          {
29              if (graph == null) continue;
30              foreach (RoomNodeSO node in graph.roomNodeList)
31              {
32                  if (node == null) continue;
33                  // Validation logic for room node types
34              }
35          }
36      }
37  #endif
38  }
```

Listing 67: Core functionality of DungeonLevelSO
Source: own elaboration

5.17.1 Drag Canvas and Draw Canvas Grid

The Drag Canvas and Draw Canvas Grid system is mainly for visualizing and designing dungeons (see Fig. 60). It enables developers to interact with a grid system that supports smooth navigation and precise element placement, which boost development progress.

The grid was implemented by calculating vertical and horizontal lines based on screen dimensions and grid size **Listing 68**. A dynamic resizing feature ensured that the grid adapted to canvas scrolling and resizing. In addition, customization options were provided to adjust grid size and opacity to suit different design needs (see Fig. 61).

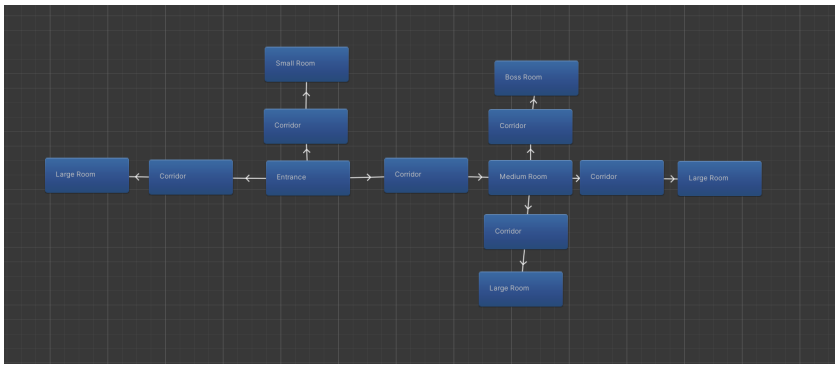


Fig. 60: Example view of level in canvas grid
Source: own elaboration

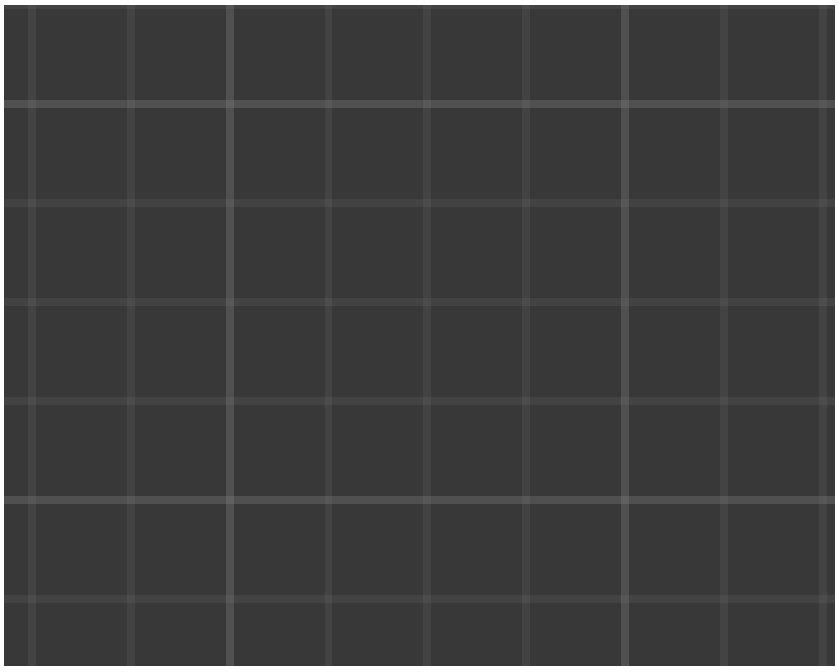


Fig. 61: Example view of grid inside canvas grid
Source: own elaboration

```
private void DrawBackgroundGrid(float gridSize, float gridOpacity, Color
    gridColor)
   {
      int verticalLineCount = Mathf.CeilToInt((position.width + gridSize)
    / gridSize);
      int horizontalLineCount = Mathf.CeilToInt((position.height +
    gridSize) / gridSize);
      Handles.color = new Color(gridColor.r, gridColor.g, gridColor.b,
    gridOpacity);
      _graphOffset += _graphDrag * 0.5f;
      Vector3 gridOffset = new Vector3(_graphOffset.x % gridSize,
    _graphOffset.y % gridSize, 0);

      for (int i = 0; i < verticalLineCount; i++)
      {
         Handles.DrawLine(new Vector3(gridSize * i, -gridSize, 0) +
    gridOffset, new Vector3(gridSize * i, position.height + gridSize, 0f) +
     gridOffset);
      }

      for (int i = 0; i < horizontalLineCount; i++)
      {
         Handles.DrawLine(new Vector3(-gridSize, gridSize * i, 0) +
    gridOffset, new Vector3(position.width + gridSize, gridSize * i, 0f) +
    gridOffset);
      }
      Handles.color = Color.white;
   }
```

Listing 68: Grid Line Calculation
Source: own elaboration

5.17.2 Layers and Sorting Layers

Layers and sorting layers are used to control rendering and interactions in Unity. Layers help define which game objects interact or collide while sorting layers determine the rendering order of overlapping sprites.

Collision management was configured using Unity's Layer Collision Matrix to define interactions between specific layers. Sorting layers were assigned to sprites, with higher values ensuring that objects were rendered on top (see Fig. 62). All layers were applied programmatically to improve the interactions between game objects **Listing 69**.

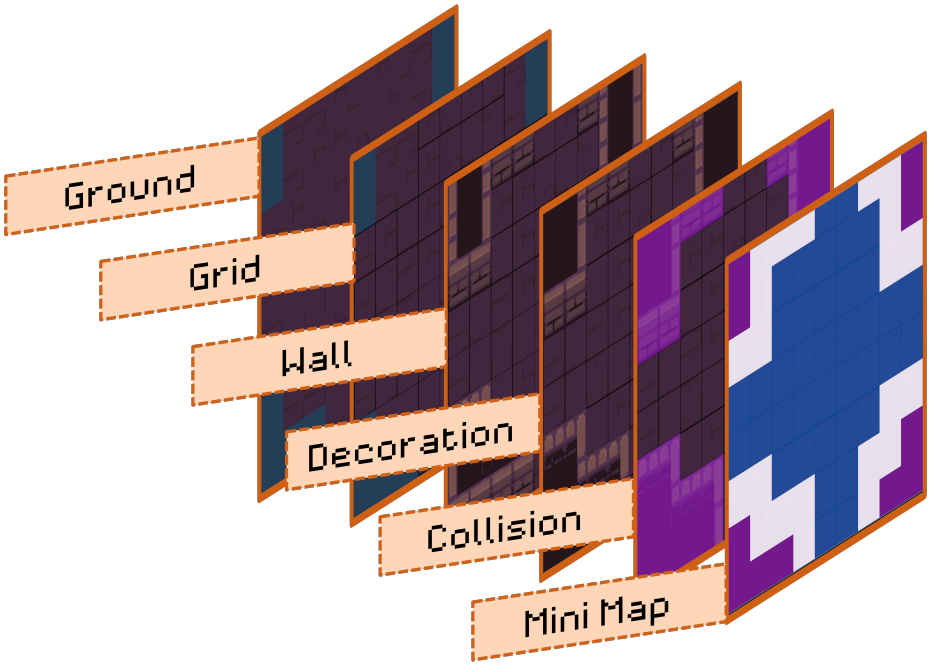


Fig. 62: Hierarchy view of Tilemaps in our project
Source: own elaboration

```
private Room CreateRoomFromTemplate(RoomTemplateSO roomTemplate,
    RoomNodeSO roomNode)
    {
        Room room = new Room();

        room.TemplateID = roomTemplate.guid;
        room.Id = roomNode.ID;
        room.Prefab = roomTemplate.prefab;
        room.RoomNodeType = roomTemplate.RoomNodeType;
        room.LowerBounds = roomTemplate.LowerBounds;
        room.UpperBounds = roomTemplate.UpperBounds;
        room.SpawnPositionArray = roomTemplate.SpawnPositionArray;
        room.TemplateLowerBounds = roomTemplate.LowerBounds;
        room.TemplateUpperBounds = roomTemplate.UpperBounds;

        room.ChildRoomIdList = CopyStringList(roomNode.childRoomNodeIDList
    );
        room.DoorWayList = CopyDoorwayList(roomTemplate.DoorwayList);
```

```
17
18          if (roomNode.parentRoomNodeIDList.Count == 0) // Enterance
19          {
20              room.ParentRoomId = "";
21              room.IsPreviousVisited = true;
22
23              GameManager.Instance.SetCurrentRoom(room);
24          }
25          else
26          {
27              room.ParentRoomId = roomNode.parentRoomNodeIDList[0];
28          }
29
30          return room;
31      }
```

Listing 69: Creating Room with Layer Masks And Local Coordinates
Source: own elaboration

5.17.3 Tilemaps, Targeted Brushes, and Advanced Palettes

The Tilemaps system helps with 2D game design by focusing on creating specialized brushes for efficient and flexible painting. This includes tools that simplify tile placement, manage palettes, and allow dynamic adjustments to get grid coordinates (see Fig. 63).

Custom brushes were created for precise placement of tiles based on specific coordinates data, as well as exporting sets of coordinates for room placement in the future. They were designed to handle specific tile sets with precision **Listing 70**. Additionally, those dynamic painting techniques were integrated, to modify tiles across multiple layers and z-coordinates.

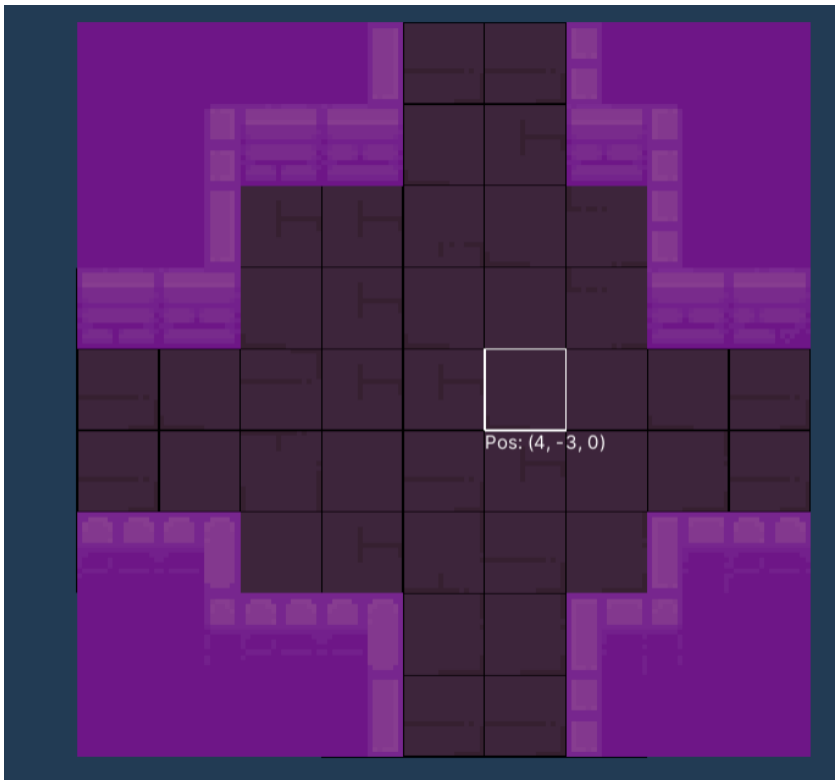


Fig. 63: Tilemap grid with a custom coordinate brush in action
Source: own elaboration

```
[CustomEditor(typeof(CoordinateBrush))]
    public class CoordinateBrushEditor : GridBrushEditor
    {
        private CoordinateBrush Brush => target as CoordinateBrush;

        public override void OnPaintSceneGUI(GridLayout grid, GameObject brushTarget, BoundsInt position, GridBrushBase.Tool tool, bool executing)
        {
            base.OnPaintSceneGUI(grid, brushTarget, position, tool, executing);

            var zPosition = new Vector3Int(position.min.x, position.min.y, Brush.zLayer);
            BoundsInt adjustedBounds = new BoundsInt(zPosition, position.size);

```

```
13              Vector3[] cellCorners = new Vector3[]
14              {
15                  grid.CellToLocal(new Vector3Int(adjustedBounds.min.x,
    adjustedBounds.min.y, adjustedBounds.min.z)),
16                  grid.CellToLocal(new Vector3Int(adjustedBounds.max.x,
    adjustedBounds.min.y, adjustedBounds.min.z)),
17                  grid.CellToLocal(new Vector3Int(adjustedBounds.max.x,
    adjustedBounds.max.y, adjustedBounds.min.z)),
18                  grid.CellToLocal(new Vector3Int(adjustedBounds.min.x,
    adjustedBounds.max.y, adjustedBounds.min.z))
19              };
20
21              Handles.color = Color.blue;
22              for (int i = 0; i < cellCorners.Length; i++)
23              {
24                  Handles.DrawLine(cellCorners[i], cellCorners[(i + 1) %
    cellCorners.Length]);
25              }
26
27              Handles.Label(grid.CellToWorld(zPosition), $"Pos: {zPosition}
    Size: {position.size}", new GUIStyle { normal = { textColor = Color.
    white } });
28          }
29      }
```

Listing 70: Custom Coordinate Brush Implementation
Source: own elaboration

One of use Coordinate Brush ordinates in **Listing 69**

5.17.4 Doorway Management

Doorways are critical for connecting rooms and ensuring smooth navigation in a dungeon. They must be positioned accurately and managed dynamically to maintain the dungeon's structure.

All doorways positions were stored as Vector2 coordinates, representing their placement inside the room. The dimensions of the doorways were recorded to help in dynamic blocking of unused doorways. Then, all tiles are placed to seal off unconnected doorways, enhancing the layout's functionality **Listing 71**.

```
private Doorway GetOppositeDoorway(Doorway parentDoorway, List<Doorway>
   doorWayList)
    {
        foreach (Doorway doorwayToCheck in doorWayList)
        {
            if (parentDoorway.Orientation == Orientation.east &&
   doorwayToCheck.Orientation == Orientation.west)
            {
                return doorwayToCheck;
            }
            else if (parentDoorway.Orientation == Orientation.west &&
   doorwayToCheck.Orientation == Orientation.east)
            {
                return doorwayToCheck;
            }
            else if (parentDoorway.Orientation == Orientation.north &&
   doorwayToCheck.Orientation == Orientation.south)
            {
                return doorwayToCheck;
            }
            else if (parentDoorway.Orientation == Orientation.south &&
   doorwayToCheck.Orientation == Orientation.north)
            {
                return doorwayToCheck;
            }
        }

        return null;
    }
```

Listing 71: Blocking Unused Doorways
Source: own elaboration

5.17.5 Dungeon Builder: Placing Rooms

The Dungeon Builder uses a systematic approach to place rooms within a dungeon, ensuring they fit together without overlapping (see Fig. 64 and 65) and form a cohesive layout.

Each room has bounding box calculated for it to define its spacial boundaries. Then overlap detection algorithm ensures there is no intersections between bounding boxes to validate the placement of the rooms (see Fig. 66 and Fig. 67) **Listing 73**. Room is only added to the dungeon if there is no overlapping with any other already existing room (see Fig. 68) and after all this will adjust room **Listing 72**.

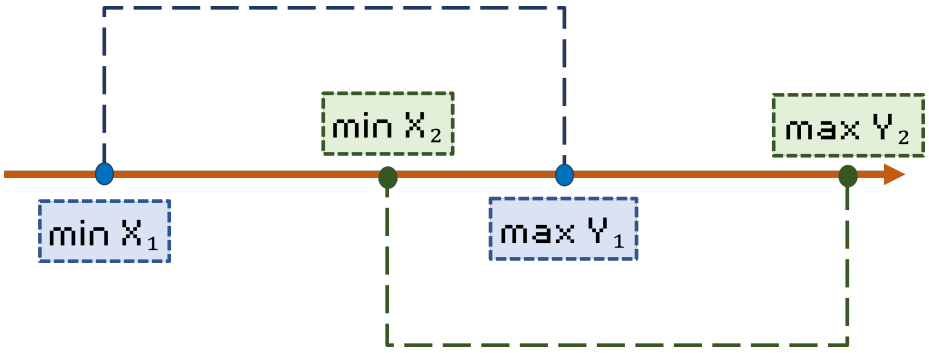


Fig. 64: Concept how to determine Overlapping Intervals in 2 points
Source: own elaboration

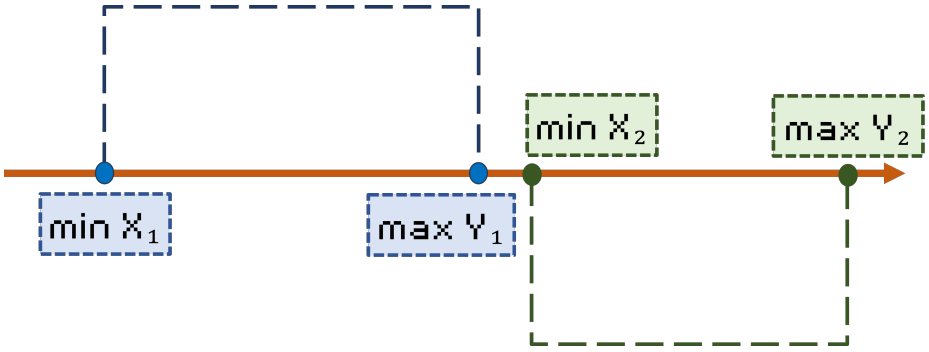


Fig. 65: Concept how to determine Non-Overlapping Intervals in 2 points
Source: own elaboration

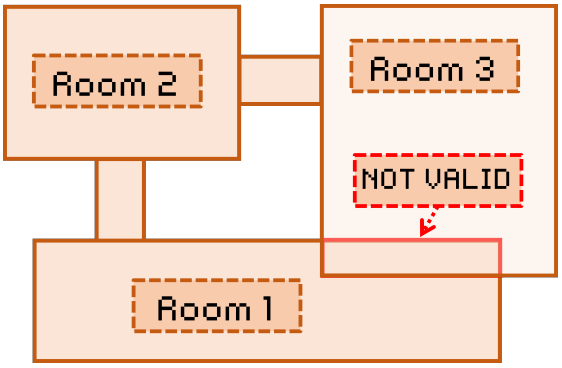


Fig. 66: Example of overlapping room in Dungeon
Source: own elaboration

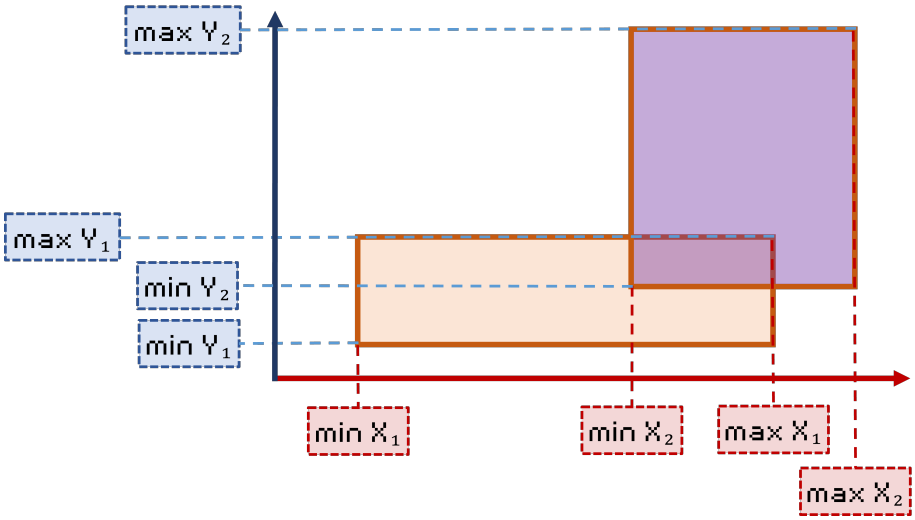


Fig. 67: Concept how to determine Overlapping Intervals with figures
Source: own elaboration

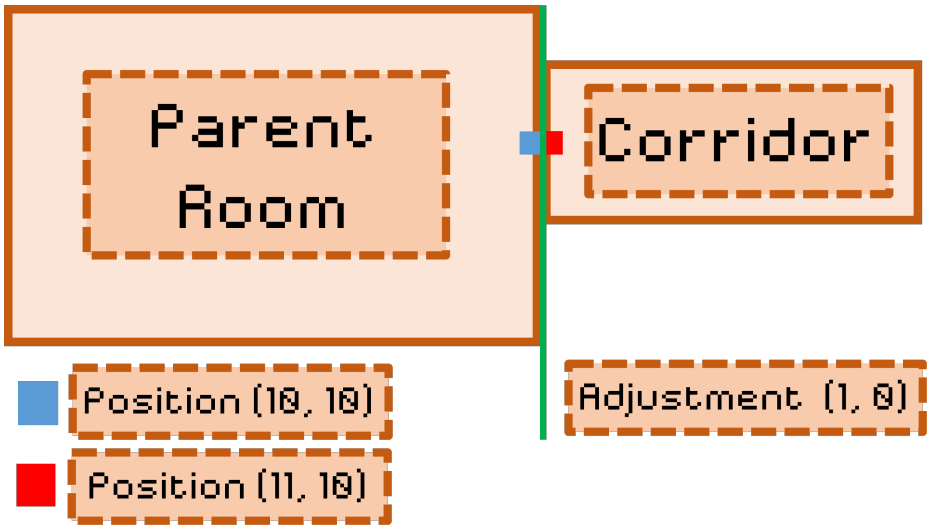


Fig. 68: Example of placing rooms in Dungeon
Source: own elaboration

```
switch (doorway.Orientation)
        {
            case Orientation.north:
                adjustment = new Vector2Int(0, -1);
                break;
            case Orientation.east:
                adjustment = new Vector2Int(-1, 0);
                break;
            case Orientation.south:
                adjustment = new Vector2Int(0, 1);
                break;
            case Orientation.west:
                adjustment = new Vector2Int(1, 0);
                break;
            case Orientation.none:
                break;

            default:
                break;
        }
```

Listing 72: Calculate adjustment position offset
Source: own elaboration

```
private bool CanPlaceRoomWithNoOverlaps(RoomNodeSO roomNode, Room
    parentRoom)
    {
        bool roomOverlaps = true;

        while (roomOverlaps)
        {
            List<Doorway> unconnectedAvailableParentDoorways =
    GetUnconnectedAvailableDoorways(parentRoom.DoorWayList).ToList();

            if (unconnectedAvailableParentDoorways.Count == 0)
            {
                return false;
            }

            Doorway doorwayParent = unconnectedAvailableParentDoorways[
    UnityEngine.Random.Range(0, unconnectedAvailableParentDoorways.Count)];

            RoomTemplateSO roomTemplate =
    GetRandomRoomTemplateForRoomConsistentWithParent(roomNode,
    doorwayParent);

            Room room = CreateRoomFromTemplate(roomTemplate, roomNode);

            if (PlaceTheRoom(parentRoom, doorwayParent, room))
```

```
            {
                roomOverlaps = false;

                room.IsPositioned = true;

                DungeonBuilderRoomDictionary.Add(room.Id, room);
            }
            else
            {
                roomOverlaps = true;
            }
        }

        return true;
    }
```

Listing 73: Overlap Detection Example
Source: own elaboration

5.18 Timers and Timer Management

The timer management system is designed to coordinate game mechanics that require timing, such as turn-based interactions and health-based actions **Listing 74** (see Fig. 69). The system maintains individual timers for players, enemies, and objects, determining priorities based on tags and ensuring smooth transitions between game states **Listing 75**.

A 'TimersManager' class was implemented to handle timer functionality, including: Adding and managing individual timers, synchronizing with in-game events through a custom 'EventSystem' (section 5.20) and supporting cheat mode for debug purposes. Each individual timer had it's entity of 'TimerData' class to encapsulate timer's properties like: tag, health bar object reference and associated text. To ensure proper game state progression timers are prioritized using a dictionary 'TagPriority' (**Listing 87**). At the beginning of each game all timers are initialized dynamically based on the type of the character it is assigned to.

Fig. 69: Example view of timers in game level
Source: own elaboration

```
void Awake()
{
    TimersManager timersManager = FindObjectOfType<TimersManager>();

    string tag = Define(_characterType);

    if (timersManager != null)
    {
        if (_characterType == CharacterType.Enemy)
        {
            int ran = Random.Range(5, 10);
            _text.text = ran.ToString();
            timersManager.AddTextFromSetTimer(_text, tag, _hp);
        }
        else
        {
            _text.text = "0";
            timersManager.AddTextFromSetTimer(_text, tag, _hp);
        }
    }
    else
    {
        Debug.LogError("Timer manager is missing!");
    }
}
```

Listing 74: Initializing a Timer
Source: own elaboration

This initialization sequence demonstrates the dynamic setup of timers, especially differentiating between enemy and non-enemy character types. The random timing for enemies introduces variability, which enhances gameplay dynamics.

```
void CalculatePriority()
{
    int highestPriority = -1;
    int highestPriorityIndex = -1;

    for (int i = 0; i < _timers.Count; i++)
    {
        string currentTag = _timers[i].Tag;
        int currentValue = _timers[i].Value;

        if (TagPriority.TryGetValue(currentTag, out int currentPriority))
        {
            if (currentPriority > highestPriority && currentValue == 0)
            {
                highestPriority = currentPriority;
                highestPriorityIndex = i;
            }
        }
    }

    _activeTimerIndex = highestPriorityIndex != -1 ? highestPriorityIndex
    : 0;
    UpdateTexts();
}
```

Listing 75: Updating Timer Priority
Source: own elaboration

This method for updating timer priority ensures that game-critical timers, particularly those with zero value, are always addressed first. The dictionary-driven priority system ('TagPriority') enhances scalability by allowing developers to add new tags or reprioritize existing ones easily.

In addition to these examples, other methods like 'PauseTimers', 'ResetTimers', and 'RemoveTimer' further illustrate the flexibility and strength of the 'TimersManager'. These methods provide comprehensive control over the lifecycle of timers, aligning them with game events and player actions.

5.19 EventEnemy and Tutorial

The event system for enemies and tutorials utilizes Unity's ToastNotification package (section 5.19.1) (**Listing 77** and **Listing 78**) to create interactive tutorials and manage player interactions with enemies and environmental events (see Fig. 70 and Fig. 71). This system coordinates gameplay introduction, enemy encounters, and room-based progression, ensuring an engaging and informative player experience.

The tutorial system is built around the `TutorialManager` class, which manages room-based progression using the `EnterRoom` (**Listing 76**), `ProceedToNextRoom`, and `CompleteTutorial` methods. It displays relevant tutorial messages based on the player's current room and progress while tracking critical tutorial milestones, such as inventory checks and enemy defeats. To display tutorial instructions, the `TypeWriterEffect` (**Listing 79**) class creates immersive text displays with a typewriter effect. All tutorial prompts and enemy interactions are managed by `ShowCaseTutorial` and `ShowCaseEvent` classes, while the `EventSystem` (section 5.20) enhances the modularity and extensibility of the tutorial mechanics.

Fig. 70: Example of Event Enemy in Dungeon
Source: own elaboration

Fig. 71: Example of Tutorial in Dungeon
Source: own elaboration

```
private void EnterRoom(int roomIndex)
{
    switch (roomIndex)
    {
        case 0:
            MessageQueueManager.ShowMessage("Greetings, traveler! I am King Arthurth III, ruler of these lands...");
            MessageQueueManager.ShowMessage("Press 'E' to take stock of your belongings.");
            break;

        case 1:
            MessageQueueManager.ShowMessage("Here we a r e the Trap Room ...");
            MessageQueueManager.ShowMessage("LEFT-MOUSE button to select card RIGHT-MOUSE button to unselect card...");
            break;

        case 2:
            MessageQueueManager.ShowMessage("This is i t the entrance to the Tower...");
            break;
    }
}
```

Listing 76: Room Progression with Tutorial Messages
Source: own elaboration

This method, part of the `TutorialManager`, demonstrates the logic for room-based messaging, providing players with context and instructions as they progress through the tutorial.

```
public static void Show(string messageText, float timerInSeconds = -1,
    string iconName = "")
{
    Hide();

    if (timerInSeconds <= -1)
        timerInSeconds = minimumMessageTime;

    Transform message = Instantiate(messagePrefab, toastNotificationTut);
    message.gameObject.SetActive(true);

    TextMeshProUGUI text = message.Find("Text").GetComponent<
    TextMeshProUGUI>();
    UnityEngine.UI.Image timer = message.Find("Timer").GetComponent<
    UnityEngine.UI.Image>();

    text.text = messageText;
    text.alignment = messageScreenPosition == MessageScreenPosition.Center
        ? TextAlignmentOptions.Center : TextAlignmentOptions.MidlineLeft;

    timer.enabled = showTimerRender && timerInSeconds > 0;
}
```

Listing 77: Toast Notification System Highlights
Source: own elaboration

This snippet highlights the `Show` method within the ToastNotification system, responsible for dynamically displaying notifications with customizable text, duration, and optional icons.

```
void Awake()
{
    messagePrefab = _messagePrefab;
    toastNotificationTut = transform;

    minimumMessageTime = _minimumMessageTime;
    hideOnClick = _hideOnClick;
    darkTheme = _darkTheme;
    showTimerRender = _showTimerRender;
    timerDirection = _timerDirection;
    messageScreenPosition = _messageScreenPosition;

    if (toastNotificationTut.GetComponent<CanvasGroup>())
        isCanvasGroup = true;
}
```

Listing 78: Toast Notification Object Initialization
Source: own elaboration

This fragment demonstrates the initialization of the ToastNotification object, setting up essential configurations such as themes, message positioning, and visibility options.

5.19.1 Integration with ToastNotification

The integration of the ToastNotification package (section 4.1.2) ensures effortless and dynamic message rendering:

- Contextual messages are queued and displayed based on player progress (`MessageQueueManager` **Listing 76**).
- Notifications are enhanced with a typewriter effect (**Listing 79**) for immersive text display.
- Interactions are tightly linked with custom events, enabling real-time updates for tutorials and enemy events.

This system provides a solid foundation for onboarding new players, guiding them through mechanics, and enhancing engagement through interactive messages and environmental storytelling.

```
private IEnumerator Typewriter()
    {
        TMP_TextInfo textInfo = _textBox.textInfo;
        while (_currentVisibleCharactersIndex < textInfo.characterCount)
        {
            if (_currentVisibleCharactersIndex >= textInfo.characterInfo.
    Length)
                break;  // Safety check
```

```
8
9           char charakter = textInfo.characterInfo[
    _currentVisibleCharactersIndex].character;
10
11          _textBox.maxVisibleCharacters++;
12
13          if (!CurrentlySkiping &&
14              (charakter == '.' || charakter == '!' || charakter == '?'
    || charakter == ',' || charakter == ':' ||
15               charakter == ';' || charakter == '-' || charakter == '
    ' || charakter == '   ' || charakter == ' ę ' ||
16               charakter == '   ' || charakter == '   '))
17          {
18              yield return _interpunctationDelay;
19          }
20          else
21          {
22              yield return CurrentlySkiping ? _skipDelay : _simpleDelay;
23          }
24
25          _currentVisibleCharactersIndex++;
26
27          if (_currentVisibleCharactersIndex == textInfo.characterCount
    - 1)
28          {
29              EventSystem.SkipedText.Invoke(true);
30          }
31      }
32  }
```

Listing 79: How Typewriter effect looks like
Source: own elaboration

5.20 EventSystem Overview

The EventSystem is the main core of interactivity and communication in the application, designed to centralize event handling and simplify interactions across various game mechanics. Built on Unity's UnityEvent class, the EventSystem promotes modularity and reduces connections between components, allowing for scalable and maintainable event-driven architecture **Listing 80**.

The EventSystem is implemented as a static class and works as a global hub for managing custom events. It represents UnityEvent instances for specific actions such as player movement, enemy turns, and environmental interactions. It organizes and enhances readabil-

ity by defining clear regions for various event types, which are utilized by components across different layers, ensuring a single source of truth for event handling. Each event declaration follows consistent naming rules, making the system intuitive for developers to extend.

```
public static class EventSystem
{
    #region MovePlayer: EnemyCotrollerTests.cs -> Timer.cs

    public static UnityEvent<int> FinishEnemyTurn = new UnityEvent<int>();

    #endregion

    /*
    ...
    */

    #region WhatHP: Timer.cs -> HealthBar.cs

        public static UnityEvent<GameObject, int> WhatHP = new UnityEvent<
    GameObject, int>();

    #endregion
    /*
    ...
    */

}
```

Listing 80: How EventSystem looks like
Source: own elaboration

The EventSystem class provides a centralized structure for event declarations, ensuring consistency and visibility across the game's event-driven mechanics.

5.20.1 Benefits and Future Improvements

- Modularity: The centralized structure allows components to interact without direct dependencies.
- Scalability: Adding new events is straightforward, ensuring future game features integrate seamlessly.
- Extensibility: Custom UnityEvent types can support complex data transfer between components.
- Potential Improvements:

- Implementing debug tools for tracking active listeners and event calls in real time.
- Refactoring unused events to maintain code base cleanliness.

5.21 GameManager

The GameManager is the central system for managing the game's state, player interactions, and level progression. It provides the infrastructure for seamless gameplay, handling everything from dungeon level initialization to player spawn logic. This ensures the game runs smoothly while maintaining flexibility for future expansions.

GameManager implements state management (section 4.9.1) to track the current game state to produce different stages like gameplay, boss battles, or level transitions (**Listing 81**). Dungeon management is handled through methods like PlayDungeonLevel and ChangeLevel, which initialize and manage dungeon levels (**Listing 82**). Player integration links the player movement, his spawn points, and interaction with dungeon overlays (section 5.17.2). In addition event integration utilizes the EventSystem (section 5.20) to respond to game-level events such as level initialization and state changes.

```
[DisallowMultipleComponent]
public class GameManager : SingletonMonobehaviour<GameManager>
{
    /*
    ...
    */
    private Room currentRoom;
    private Room previousRoom;
    [HideInInspector] public GameState GameState;

    private void Start()
    {
        GameState = GameState.gameStarted;
        _playerController = _player.GetComponent<PlayerController>();
        EventSystem.NewLevel.AddListener(ChangeGameState_NewLevel);
    }
    /*
    ...
    */
private void HandleGameStates()
{
    switch (GameState)
```

```
    {
        case GameState.gameStarted:
            PlayDungeonLevel(currenDungeonLevelListIndex);

            GameState = GameState.playingLevel;
            break;

        case GameState.bossStage:
            if (currenDungeonLevelListIndex < dungeonLevelList.Count - 1)
            {
                ChangeLevel();
                PlayDungeonLevel(currenDungeonLevelListIndex);
                GameState = GameState.playingLevel;
                //...
            }

        case GameState.gameLost:
        //...
    }
}
```

Listing 81: Handling Game States
Source: own elaboration

This fragment highlights the core logic for transitioning between different game states, ensuring that dungeon levels are initialized or progressed seamlessly.

```
private async Task PlayDungeonLevel(int dungeonLevelListIndex)
{
    EventSystem.NewLevel.Invoke();
    bool dungeonBuiltSuccessfully = DungeonBuilder.Instance.
    GenerateDungeon(dungeonLevelList[dungeonLevelListIndex]);

    if (!dungeonBuiltSuccessfully)
    {
        Debug.LogError("Failed to build dungeon");
        return;
    }

    _player.transform.position = HelperUtilities.
    GetSpawnPositionNearestToPlayer(_player.transform.position);
}
```

Listing 82: Dungeon Level Initialization
Source: own elaboration

This method showcases how dungeon levels are built and how the player is positioned within the generated environment.

5.22 GameResources

The GameResources class is responsible for managing and providing global game assets and configurations. It works as a central repository for dungeon-related resources and reusable materials, ensuring consistency and efficient resource access across the game.

The system uses the singleton pattern (section 4.9.2) to ensure a single instance of GameResources that is globally accessible via the Instance property (**Listing 83**). It serves as a container for critical resources like room node types and reusable materials. It also utilizes Unity's Resources.Load for flexible and efficient asset management (**Listing 84**).

```
public static GameResources Instance
{
    get
    {
        if (_instance == null)
        {
            _instance = Resources.Load<GameResources>("GameResources");
        }
        return _instance;
    }
}
```

Listing 83: Singleton Implementation for GameResources
Source: own elaboration

This singleton implementation guarantees that GameResources is instantiated once and provides a global access point for resources.

```
public RoomNodeTypeListSO roomNodeTypeList;

// Example usage
RoomNodeTypeListSO nodeList = GameResources.Instance.roomNodeTypeList;
Debug.Log($"Room Node Types Loaded: {nodeList.Count}");
```

Listing 84: Accessing Room Node Types
Source: own elaboration

This example illustrates how game components can easily access room node types for dungeon-related operations.

5.23 Enums

The Enums file contains all the enumerations used across the project, centralizing definitions for consistent use. Each enum provides a well-defined set of values representing states, types, or configurations for various gameplay elements. This approach improves code readability and maintainability while preventing magic numbers or ambiguous states in the system.

It includes key elements such as 'Orientation' which specifies cardinal directions for corridor alignment and navigation, 'CharacterType' (**Listing 87**) which categorizes entities like players, enemies, and items for timers and event handling. 'GameState' (**Listing 85**) defines different stages of the game life cycle, enabling efficient state management and transitions. 'CardType' classifies cards used in gameplay mechanics, such as attack, defence, and movement cards. Lastly, 'HitAnimation' (**Listing 86**) defines visual effects triggered during hit animations for immersive feedback.

```
1  public enum GameState
2  {
3      gameStarted,
4      playingLevel,
5      engagingEnemies,
6      bossStage,
7      engagingBoss,
8      levelCompleted,
9      gameWon,
10     gameLost,
11     gamePaused,
12     dngeonOverviewMap,
13     restartGame
14 }
```

Listing 85: Enum for Game States
Source: own elaboration

This enum ensures a well-structured representation of the game's state, allowing for seamless transitions and robust state management.

```
public enum HitAnimation
{
    Arrow,
    Blood,
    Boom,
    Fire_Meteor,
    Electric,
    Physical
}
```

Listing 86: Enum for Hit Animations
Source: own elaboration

This enumeration simplifies the management of visual effects, enabling developers to easily integrate or extend animations for different attack types.

```
public enum CharacterType
{
    Player,
    Enemy,
    Item,
    Boss,
    None
}
```

Listing 87: Enum for Character Types in Timer Events
Source: own elaboration

This enum categorizes game entities for use in systems like timers or interactions, ensuring consistent handling across various gameplay components.

5.24 HealthBar

The HealthBar class manages all health-related functionality for linked GameObjects. It provides a visual representation of health and integrates with the event system to respond to changes such as damage, healing, or destruction of the associated entity. This system ensures that health-related mechanics are intuitive and seamlessly integrated into the gameplay (see Fig. 72).

The system tracks and updates the current and maximum health values and reflects the changes visually using a slider and textual representation (**Listing 89**). By receiving invokes through the 'EventSystem' (section 5.20) it changes the hp value and in a case of health

value becoming less or equal to zero it handles destruction and removal of the associated game object (**Listing 90**). All this is dynamically linked with the assigned health bar object and is able to synchronize with the current game state and room to keep all events accurate (**Listing 90**).

Fig. 72: Example view of HP System in game
Source: own elaboration

```
private void Start()
{
    Split();
    SetData();
    UpdateHealthText();
    EventSystem.AssignTimerIndex.AddListener(SetTimerIndex);
    EventSystem.WhatHP.AddListener(HandleWhatHP);
}
```

Listing 88: Initialization of HealthBar
Source: own elaboration

This fragment demonstrates the initialization process, linking the health bar to the event system and preparing its initial state.

```
public void ChangeHealth(int health)
{
    _value += health;
    _value = Mathf.Clamp(_value, 0, _maxValue);
    _slider.value = _value;
    UpdateHealthText();
}

private void UpdateHealthText()
{
    _data.text = $"{_value}/{_maxValue}";
}
```

Listing 89: Health Change and Update
Source: own elaboration

This code highlights how the health bar visually updates to reflect changes in health, ensuring accurate feedback for the player.

```
private void Kill()
{
    if (_value <= 0)
    {
        if (gameObject.CompareTag("Player"))
        {
            EventSystem.OpenGameover?.Invoke();
            return;
        }

        Debug.Log("killed: " + myTimerIndex);

        if (room != null)
        {
            room.enemyInRoomList.Remove(this.gameObject);
        }
```

```
17
18          EventSystem.DeleteReference.Invoke(_timerNumbToDelete);
19          Destroy(_body);
20      }
21
22  }
```

Listing 90: Handling Object Destruction
Source: own elaboration

This code demonstrates how the system handles entity destruction when health is depleted, including removing references and invoking relevant events.

5.25 Testing results

5.25.1 Prototype Completion (April 10)

Testers were complaining about: the game being more like template fo it than actually something, they didn't like movement inputs being on arrows, player's character could walk through walls, game mechanics weren't connected with the events happening in game, so they had to start them either by an input on a keyboard or by pressing the button on the screen, game cards responsible for players abilities were more like a visual effect than a feature, inventory wasn't coherent, items in inventory weren't following mouse cursor directly, enemy didn't do anything apart of chasing a player, while chasing the player enemy was sometimes cutting corners instead of moving by the grid, main menu was only a place holder.
Testers liked: the idea for the game, working path finding, the cards cost system, dungeons visuals were nice looking
Summary: There was still a lot of work to do and working scenes aren't the best to showcase how does something work in game.

Final feedback:

- Only movement was a bringing any change in a game
- A lot of binds to keyboard instead of proper game mechanics depending on each other
- Cards were only a visual. They weren't bringing any change
- Character could walk through walls
- Path finding was pretty impressive

5.25.2 Alpha Release (May 8)

Testers were complaining about: no information about what's happening in the game, some of the mechanics still were only available by using a manual key bind, no interaction with the grid on the floor of the map, no information who can be attacked, walking through walls was still possible in some places, cards were only dealing damage even if they were healing cards, card abilities didn't have range restriction, enemy wasn't doing anything apart of standing and receiving damage, items in inventory weren't following mouse cursor directly.
Testers liked: that there was some interaction between the player and the enemy, cards weren't just a visual effect, movement was more comfortable by moving it from arrows to WASD buttons.
Summary: There was progress since first testing session, there was still a lot of work to do, but some of it was already in progress during the testing session.

Final feedback:

- Pause menu was missing on one of the scenes
- After creating a deck for the first time player weren't able to change.
- Not enough information on what to do and what's happening

5.25.3 Beta Release (May 29)

Testers were complaining about: not many game play changes comparing to previous testing session, possibility of running out of cards during the battle, no actions from enemy's side, no mouse interaction with the grid of the floor, no information who can be attacked, card abilities didn't have range restriction.
Testers liked: walking through walls wasn't possible anymore.
Summary: Not many changes due to the spring break and a lot of other projects during the semester. We received crucial information to put much more work in visual information about what's happening in game.

Final feedback:

- Game was impossible to beat

- Not enough information on what's happening in game
- Not enough information

5.25.4 Final Release (June 19)

Testers were complaining about: not existing ending of the game, no mouse interaction with the grid of the floor, no information who can be attacked, card abilities didn't have range restriction, enemies were moving at random and didn't pose any threat, no balance in cards drawing, movement cards didn't affect movement .
Testers liked: existing game play loop, deck refilling after running out of cards, indication of received and healed damage by game entities, combat and deck editing mode, working inventory that shows how the deck looks like depending of items equipped, graphical changes in how map looks like, addition of mini map.
Summary: A lot of progress in game play mechanics, the game needs much more visual effects, more balance of how cards are drew from the deck and how much they heal or deal damage.

Final feedback:

- Almost none visual effects that tell the player his combat started
- Apart from movement rest of actions should have button on the screen

6 Discussion

This section describes the lessons learned during the completion of each system included in the project. It outlines how the design process affected the elements of design and why it happened.

In every project, there is some room for improvement. The same applies to this one. Some parts get omitted due to time constraints, some are simply things that developers did not come up with, and some simply did not work. In this section, we list those parts and explain how they could influence the end result of our project.

6.1 Cooperation

Many modern games include a game mode that lets players play together in a multiplayer or a cooperation game mode. This becomes easier to implement thanks to the game engine site implementations of multiplayer interfaces. Some platforms, similar to "Steam", give game developers the ability to completely avoid many parts of those systems' implementation by letting players connect to the same game from different places via a simulated LAN connection. In this game, the implementation of a cooperation system would require an update to the balancing of cards and enemy difficulty. For example, the more players are in the game, the more health and/or damage they receive and do. This mechanic could also allow players to go through a dungeon quicker, by exploring many rooms at the same time, which would require developers to create more complicated levels, the more players there are in the game.

6.2 Creating Items

A fun addition to the game could be the ability for a player to create his own items. This could be achieved by first letting him destroy currently owned ones, to cards or special type of item called "item parts". From those cards or "item parts" then at a player spawn position, could stand an item crafting place (for example: an anvil), in which the player could create a custom item. This mechanic brings a couple of problems to resolve. One of them being, that if after destroying an item, the player is given cards, how do we determine what cards he gets? Maybe a good idea would be to give

the player a percentage chance to get a card of each quality: 50 for lowest, 35 for medium, and 15 for highest. This would ensure that the player would not be able to create items that would be too good, too soon. Another problem would be to make the game react to players' actions by adjusting its difficulty by making enemies stronger, but not strong enough to make the whole mechanic seem pointless. The last problem would be to create a system in Unity that would let us create Scriptable Objects in the game. It would require that the whole system be created from scratch since Unity does not let players create new assets during gameplay. An easy resolution to that problem would be to find an outside package that would enable us to do the same without having to make our own implementation.

6.3 Refining Narrative

Currently, the game's story takes on more of a background role. Throughout the dungeon, the player can find many characters from which he can deduce some parts of it, but these are mostly symbolic and do not let the player get into a state of immersion with the world of the game as he would if given more of it. The story could start when first time entering the game. An animation would play, and the narrator would give us a basic background of what is happening. Then when entering the game a player could meet characters that would give him parts of the story once he interacts with them. Those pieces would then connect into the greater whole that would then be able to be viewed in a special window in the main and in-game menus.

6.4 Quests

At this time, many characters can be found in the dungeon, however, their only purpose is to talk to the player and give him some background on the game lore. This might result in players skipping the dialogues completely since they do not bring anything of value to the gameplay. This can be changed by the addition of a new "Quest system". This could work for example, by spawning a special NPC. After the player talks to that NPC, a special room would appear in the dungeon, and the player would be given instructions on what

he is supposed to do. The quests could also require the player to meet that same NPC multiple times on different levels in order to complete them. Each time a quest is completed, the player would be given an item, the quality of which would be dependent on how deep into the dungeon the player has made it to.

6.5 Companions and Hub

In our game, the player can meet various characters he can interact with. Those characters, however, do not have any extensive use. That could be changed, by introducing two mechanics. Firstly, a special type of NPC could be created, that could be found throughout the dungeon. Those NPCs could give the player a special quest which completion would let the player access the NPC's mechanics. As an example, a "shopkeeper" NPC can be taken. After completing his quest the NPC would provide a new way for the player to gain more equipment. That equipment could be unobtainable in any other way. Secondly, a way to access those NPCs has to be created. As of now, there are two possible ways: a special room in the dungeon unique to the NPC, and a special level acting as a Hub where all NPCs will be able to be accessed. Continuing with the same example, our salesman could have his shop randomly positioned inside one of the dungeon corridors, have a special room free of any enemies, or be grouped with other NPCs in the Hub level having his shop next to the properties of other NPCs. Additionally, it would have to be considered, if the option of having the enemies appear inside the dungeon itself is chosen, how many can appear at the same time.

6.6 More Enemies

The game contains only one type of enemies - a melee enemy. These enemies are enough to create a challenging experience for the player, especially with larger numbers of units, but will introduce much repetitiveness into the gameplay. One solution would be to add more enemy types and classes of enemies that would vary enough in their movement patterns, that the player will be forced to change his playstyle in accordance to what enemy he encounters. Another solution could be to create various behavioral patterns for each enemy

type, which will change depending on how deep inside the dungeon the player has gotten into. These pattern variations would ensure that the player may not be able to always predict what a certain enemy would act like, making the gameplay more challenging, but at the same time potentially making the player's approach to fighting them a very safe one only, adding even more repetitiveness. All in all the right future possible approach would be to merge both of the ideas into one greater whole.

6.7 Limitations in the Save System

Many games make use of systems that let their players continue to progress after exiting it. This same system is used in this game, though not as extensively as in others. Still, there have been some hurdles during the implementation of the system that led to several problems.

At first, the system was supposed to work closely with the "Addressables" asset management system. It was supposed to make use of the ability of the user to give each asset tags, on which basis the assets then could be accessed in-game. For the save system, two tags have been created: locked and unlocked. These tags would be switched, any time the player would clear a level of a dungeon, or when a completely new game is started. However, later it turned out that there is no simple way to change those tags in the game "Build" itself, so the system had to be redesigned to a custom one, making use of a .json file where names of the unlocked items are stored in the list, together with a place in the equipment they might have been assigned to, and when the game starts those items are then found through the "Addressables" assets system, and assigned to the right place.

Another problematic place inside the save system was the "key rebinding". The system would make use of the PlayerPrefs class that in theory let the system save simple data inside a map built into the engine itself, however as it turns out, the system while working fine in the Unity editor, struggled inside the Build of the game. Because of that factor, this part of the system also had to be redesigned in a way to make use of a .json file stored in the game's files.

7 Conclusions

Throughout the completion of the project, there were many accomplishments, but also many things that did not work out as they were supposed to at first. Through this paper, all of those things were described. At first, the paper starts with an outlining of the key elements of the game that were to be achieved. These were backed by the games chosen as inspirations. Later, while completing the project, it turned out that some of the planned mechanics would not work well the way that they were planned, or would not fit the game at all. Even the parts that were fully implemented were often changed. It has given this game a chance to be different from all the games it is based on, even when it uses very similar mechanics.
The project has been completed with the use of many tools, the use of which has been described in the "Tools and Methods" section of the paper. These tools enabled the creation what needed to be created much quicker than they would be otherwise.
The completion of the project has yielded many results, all of them written down and explained in the "Results" section, through which came lessons learned. These lessons have been thoroughly described in the "Discussion" section, together with things that could have been accomplished, given more time and resources.
After the completion of the project, we think that even though not all the goals set have been achieved, and the process has been burdened with many problems, the end result came out quite satisfactory.

8 Licences

Unit-card-play

References

1. Creating unique custom card sets: A guide to card game design and self-publishing
2. Level up your programming with game programming patterns
3. Random-player games
4. Bostan, B., Tingoy, O.: Game design and gamer psychology pp. 105–121 (2016). https://doi.org/10.1007/978-3-319-29904-4_7, https://doi.org/10.1007/978-3-319-29904-4_7
5. Brazie, A.: Game feel: A beginner's guide. https://gamedesignskills.com https://gamedesignskills.com/game-design/game-feel/
6. Brändle, F., Wu, C., Schulz, E.: Leveling up fun: Learning progress, achievement, and expectations influence enjoyment in video games (03 2024). https://doi.org/10.31234/osf.io/vg8dz
7. Cook, M., Colton, S., Gow, J.: The angelina videogame design system—part i. IEEE Transactions on Computational Intelligence and AI in Games **9**(2), 192–203 (June 2017). https://doi.org/10.1109/TCIAIG.2016.2520256
8. Hunicke, R., Leblanc, M., Zubek, R.: Mda: A formal approach to game design and game research. AAAI Workshop - Technical Report **1** (01 2004), https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=http://www.cs.northwestern.edu/~hunicke/MDA.pdf&ved=2ahUKEwj77PDXje-GAxVO2SoKHZkCCaoQFnoECBIQAQ&usg=AOvVaw2etfhAVO5siwjt_9Iij_Zh
9. Kinstler, N.: Building a home: How to construct a card game. gamedeveloper.com (2018)
10. Koutonen, J., Leppänen, M.: How are agile methods and practices deployed in video game development? a survey into finnish game studios pp. 135–149 (2013)
11. Maurice Bergsma, M.B., Spronck, P.: Adaptive spatial reasoning for turn-based strategy games. Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment **4**(1), 161–166 (Sep 2021). https://doi.org/10.1609/aiide.v4i1.18690, https://ojs.aaai.org/index.php/AIIDE/article/view/18690
12. Michael Desharnais, Wilmer Lin, E.O.B.D.A.L.C.N., Valoroso, S.: User interface design and implementation in unity. https://unity.com (2021), https://unity.com/resources/user-interface-design-and-implementation-in-unity
13. Shaker, N., Togelius, J., Nelson, M.J.: Procedural content generation in games (2016)
14. Silva, C.G.P.d.: Analysis and development of a game of roguelike genre. Lume Home p. 76 (2015), https://lume.ufrgs.br/handle/10183/126085

List of Figures

List of Tables

Listings