Joint Drops – Creation of a 3D cooperative puzzle-solving multiplayer video game

Érica Masmano Fons

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

May 22, 2023

Supervised by: Sandra Catalán Pallarés

To my parents,
whom I admire for their strength and determination.

Acknowledgments

First of all, I would like to thank my Final Degree Work supervisor, Sandra Catalán Pallarés, for her support and dedication to my project. Her enthusiastic and attentive listening every time I shared my game's progress was invaluable, and her keen interest in its development kept me highly motivated and enthusiastic throughout its preparation.

Additionally, I would like to express my deep gratitude to my dear family - my father, mother, and grandmother, as well as my second little family - Gon and Rengar. Their consistent encouragement, unwavering support, and faith in my abilities have been a tremendous source of inspiration for me and have played an integral role in my success.

Moreover, I would like to take this opportunity to express my profound gratitude to the Unity community, and especially to Code Monkey, for their remarkable efforts in sharing top-notch knowledge, accessible to all, on a non-profit basis. Their dedication and benevolence have been instrumental in enabling me to enhance my programming skills, and have played a significant role in my growth and development as a game developer.

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

Abstract

This document presents the technical report of the Final Degree Project 'Joint Drops'. The project involves the development of a small-scale multiplayer video game that offers cooperative puzzles, wherein the players' interaction play a crucial role in overcoming each level.

The purpose of this game is to distinguish itself from the others, as multiplayer games have become a highly successful genre due to their ability to bring people together and provide a fun experience. Currently, there is a noticeable shortage of cooperative and enjoyable games in the video game industry, making this an opportune time to take advantage of the market's need for such games.

Nevertheless, the driving force behind this project is the desire to bring to life a conceptual idea that originated from another subject within the degree program. At the same time, the project aims to enhance the skills of creating clean and scalable code, while also paying attention to the creative aspects of the project.

Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

Contents

The following paragraphs present a comprehensive rationale for the creation of the proposed video game, outlining its purpose and defining a set of objectives that must be fulfilled to validate its development.

1.1 Work Motivation

The present work has been motivated by the desire to bring together the knowledge acquired and developed during the degree, including subjects related to art, narrative, design, and programming. Despite covering a broad range of topics, one area that has been underrepresented is the implementation of multiplayer features, particularly for online games.

Multiplayer games have become an increasingly prominent part of our society [9], as they tap into the innate human desire to compete and interact with others, either in a friendly way or in a truly competitive way. However, until recently, Unity did not provide a comprehensive solution for the development of multiplayer games.

The long-awaited Unity solution, *Netcode for Gameobjects* [19], was announced and it has addressed this gap in the market, making it easier for developers to implement

multiplayer features within the Unity engine. The full release version (1.1.0) became available on October 10, 2022, presenting a valuable opportunity for programmers to develop their skills in this area.

Overall, the development of a multiplayer game, using this new tool, presented an excellent opportunity to take on a challenging project that would contribute to the final degree work. Additionally, this project offered the potential to enhance programming and modeling skills.

1.2 Objectives

This section is crucial to provide a clear understanding of the project's goals and objectives. These are presented below in descending order of significance:

1. **Develop a game in Unity**: This objective involves using the Unity game engine to create a complete and functional game. This may include designing and implementing the game mechanics, creating the game assets, and programming the game functionality using Unity's scripting language, C#. The focus of this objective is to gain experience in game development using Unity and to demonstrate proficiency in creating a polished, playable game.

2. **Online Multiplayer**: The objective of implementing online multiplayer is to allow players to connect and interact with each other in a shared virtual environment, enabling a more engaging and immersive experience for players who are unable to physically play together.

3. **Good Programming practices**: The objective of clean and scalable code programming is to create high-quality, maintainable and efficient code that can be easily modified, updated and extended as the project evolves. This will ensure that the codebase is easy to work with, making it easier to identify and fix bugs, and add new features in the future.

4. **Stylized 3D modeling**: The objective of stylized 3D modeling is to create visually appealing and unique 3D assets that fit the aesthetic of the game. This will involve designing and creating models, textures and other visual elements to convey the desired style and atmosphere of the game.

5. **Level Desing**: The objective of level design is to create game levels that are both challenging and engaging for players. This involves designing the layout, structure, and mechanics of the game levels, as well as placing objects and other interactive elements.

6. **Efficient organization**: The objective of efficient organization is to ensure that the development process is well-organized and streamlined, allowing to work efficiently and effectively. This includes managing tasks, schedules, and resources.

1.3 Environment and Initial State

The project takes the conceptualized project made for the Video game Conceptual Design (VJ1222) subject as an starting point. The project involved creating a Game Design Document for a game that currently has undergone significant modifications compared to its original version, while yet still preserving its essence.

As the release of Netcode for Gameobjects played a significant role in motivating the arrange of the project, no other development tools were considered. However, the tool required an extensive exploration of documentation and resources to acquire the necessary knowledge to effectively use it before starting with the project.

To maintain code consistency, a set of naming conventions were established. Constants are named using the 'UpperCase SnakeCase' convention, while properties, events, and function names are named using the 'PascalCase' convention. Fields and function parameters, on the other hand, are named using the 'camelCase' convention.

With that in mind, the work has been carried out individually, with a focus on minimizing organizational tasks and maximizing attention and resources on essential aspects. Approaching the project on my own helped to avoid any potential creative conflicts or communication issues, leading to a smoother workflow.

1.4 Additional information

1.4.1 Key words

Video game, cooperative, online multiplayer, 3D, puzzles, Unity Netcode for Gameobjects

1.4.2 Related subjects

While most of the subjects covered in the degree have been relevant to the project, certain ones have proven to be indispensable for the successful completion of the work. The most significant aspects of each of them are detailed below:

1. **Programming II (VJ1208)**: The course provided a comprehensive introduction to object-oriented programming with the C# language, which is widely used by the Unity game engine.

2. **Game Engines (VJ1227)**: The course provided a deeper understanding of the technical aspects of game engines, with a particular emphasis on Unity, along with techniques for optimizing and improving performance.

3. **Multiplayer Systems and Networks (VJ1228)**: The course provided an introduction to different multiplayer architectures, such as client-server and peer-to-

peer, and their respective advantages and disadvantages. Additionally, it covered essential network protocols and technologies used in multiplayer games, including TCP and UDP.

4. **Graphics Communication (VJ1212)**: The course proved to be extremely valuable in the development of the game's visual assets. With the basic understanding of Blender's features, it was learned to create complex 3D models, apply textures, and render objects and environments with high quality. This enhanced the overall aesthetics of the game, providing a visually appealing experience to the players.

5. **Software Engineering (VJ1224)**: The course imparted effective work methodologies for better project organization, resulting in enhanced productivity. Additionally, it proved to be highly beneficial in terms of analyzing system requirements and preparing the relevant diagrams.

6. **Video game Conceptual Design (VJ1222)**: The course not only imparted valuable knowledge for designing levels, objectives, worlds, and mechanics, as well as gamification but also facilitated the use of methodologies that can expedite game development considerably, including the creation of a game design document.

CHAPTER 2

Planning and resources evaluation

Contents

This chapter outlines the planning and the necessary resources used in the development of the project.

2.1 Planning

In this project, the SCRUM methodology has been used to plan and manage the development process. This is an Agile project management framework used for software development that involves breaking down the project into smaller parts called sprints (see Figure 2.1).

The following section presents a detailed timeline for the project, including all tasks, subtasks, and their interdependencies. The initial technical proposal's planning was followed, with the addition of an extra research and documentation phase required for the development of the multiplayer component. As anticipated, the programming aspect of the project took precedence over the artistic component as it follows:

- **Game Design Document (10 hours)**: This task involves creating a detailed document that outlines the overall vision and design of the game. It includes information about the game mechanics, story, levels, characters, assets, and other important details. The purpose of this document is to provide a clear and concise guide for the rest of the development process.

- **Programming Game Core (90 hours)**: This task involves developing the core functionality of the game. It includes programming the game mechanics, levels, menus, user interface, and other essential features that make up the game. This is a critical task, as it lays the foundation for the rest of the development process.
  - Player actions and gamepad feature (30 hours)
  - Emote wheel (5 hours)
  - Collectibles, Interactables, Activables (10 hours)
  - Items, Special Objects (20h)
  - Game Screens adn Head-Up Display (20h)
  - Game flow Manager, Health system, Sound and Music system (5 hours)
- **Documentation and Research (40 hours)**: This task refers to the process of researching and learning about the new Unity tool that is used for developing multiplayer functionality in the game. The reason why this task may take longer than expected is that the tool is a recent release, which means that there may not be a lot of documentation or resources available yet.
- **Programming Multiplayer System (60 hours)**: This task involves developing the multiplayer functionality of the game. This can include synchronizing network objects, including a lobby system, and allocating servers with Relay. It may also require some programming code rework. Multiplayer functionality is essential and it requires a significant amount of time.
- **3D Modelling and Animation (40 hours)**: This task involves creating the 3D models and animations for the game. This includes designing and modeling the game assets, creating animations for characters and objects, and importing these assets into the game engine. This is an important task for creating a visually appealing game.
- **Stylish the game (30 hours)**: This task involves adding the finishing touches to the game's visual and audio aspects. This includes composing the game's assets, designing shaders for visual effects, adding sound effects and music, and creating visual effects. Styling is an important task that can greatly enhance the player's experience.
- **Game feel, Optimization, and Testing (20 hours)**: This task involves polishing the game to ensure it is fun to play and runs smoothly on different platforms. This includes fine-tuning game mechanics to improve the overall feel of the game, optimizing the game's performance, and testing the game to identify and fix any bugs or issues.
- **Final report and Presentation (40 hours)**: This task involves documenting the entire development process as well as the presentation.

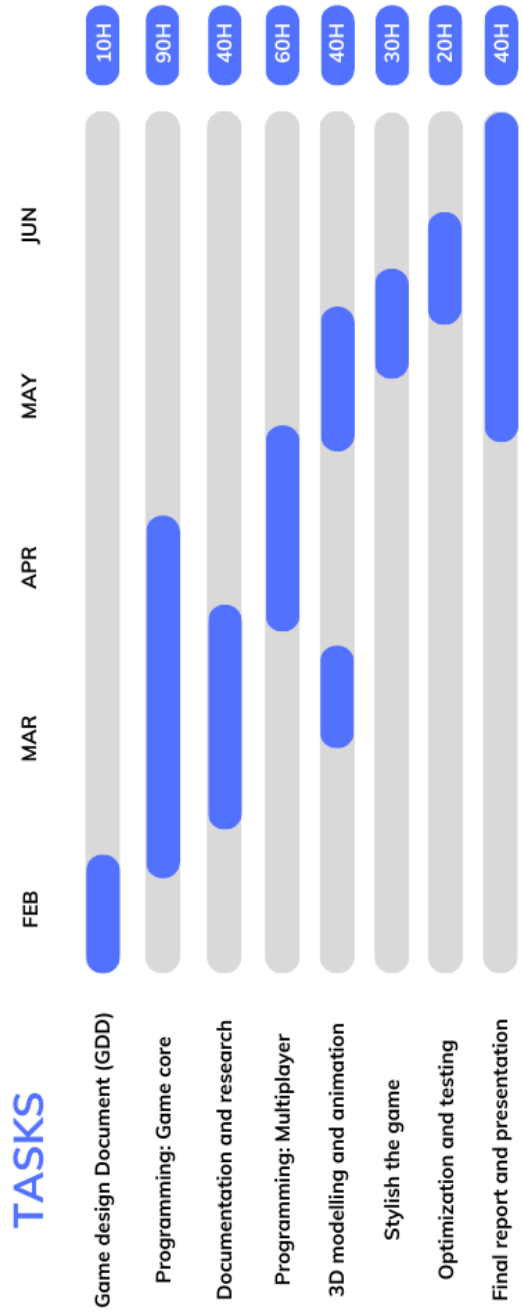

Figure 2.1: Gantt chart (made with Canva) [3].

2.2 Resource Evaluation

Before starting any project, it is important to estimate the costs involved. This includes calculating both the human resources and equipment costs to assess the viability of the project. In the case of this project, the estimated cost for a junior programmer working 300 hours would be approximately €3,400 plus taxes, based on an average salary of €21,500 per year in Spain [7]. It is also important to consider the cost of necessary hardware and software below:

2.2.1 Hardware

The development of this project involves the use of different hardware, including:

- **Laptop**: Msi GL62M 7REX (€1,200)
  - **CPU**: Intel(R) Core(TM) i7-7700HQ CPU 2.80 GHz
  - **GPU**: GeForce® GTX 1050 Ti with 2GB GDDR5
  - **RAM**: DDR4-2400 8 GB
  - **Storage**: M.2 SSD SATA 128GB + 1TB
  - **Cooling**: Cooling Boost 4
  - **Power supply**: 150W adapter
- **Monitor**: BenQ ZOWIE XL2411P 24" LED 144Hz (€130)
- **Mouse**: Razer DeathAdder V2 (€60)
- **Keyboard**: Newskill Suiko Ivory Mechanical Keyboard (€70)
- **Gamepad**: Sony Dual Shock 4 (€50)
- **Graphic tablet**: Huion Kamvas Pro 13 (€250)

2.2.2 Software

The development of this project involves the use of different software tools such as:

- **Unity** (2021.3.16f1): is a cross-platform game engine involving an integrated development environment (IDE). This and the following tools will be used for: Multiplayer game development [24].
  - **Netcode for Game Objects**: This tool allows for a connection to a host through its IP and port. The host must be on the same network as its clients [19].
  - **Relay**: A server that enables players to connect through the internet with a public IP that can be reached by both the host and its clients [22].

  - **Lobby**: This tool simplifies the initial connection between players by allowing them to create lobbies, share data, and find other lobbies before establishing a real-time network connection [25].

- **Github Desktop** (3.2.2): serves as an Internet hosting service for software development and version control through Git. This tool will be used for: Project back-ups [11].
- **Blender** (3.2.0): is a 3D modeling and sculpting software that is free, open source, and compatible with multiple operating systems. This tool will be used for: 3D modelling (player, environment and props), texturing and animation [2].
- **Krita** (5.0.0): is a free and open-source software for digital art that provides tools and features for creating illustrations and paintings. This tool will be used for: In-game illustrations [8].
- **Trello** (2.13.10): s a web-based application that follows the kanban-style approach for organizing lists and tasks. This tool will be used for: Tasks organization [1].
- **Canva**: is a free graphic design website that allows you to create documents of any kind with a professional result. This tool will be used for: Gantt chart creation for task planning [3].
- **Visual Studio Community 2022** (17.4.4): is an integrated development environment (IDE) created by Microsoft that provides a range of tools and helpful features for software development such as code completion, syntax highlighting, and debugging tools. This tool will be used for: Code writing [12].
- **Overleaf**: is an online collaborative tool for writing, editing, and publishing documents in LaTeX, a document preparation system used for typesetting technical and scientific documents. This tool will be used for: Final report making [13].

The estimated total cost of the project is €5,160, which includes the cost of free software, hardware expenses of €1,760, and a salary cost of €3,400.

CHAPTER 3

Game Design Document

Contents

This chapter aims to give an overview of the work, from a conceptual point of view of the game, detailing in depth its narrative design, levels, mechanics, and artistic style.

3.1 One Sheet

3.1.1 Concept Overview

Platform: PC
Target Age: 10-up
ESRB rating: Everyone 7+
Genre: Cooperative puzzle
Current status: In development

3.1.2 Game Summary

Recruit your friends to enter the walls of the dark castle. Some parasitic beings have invaded *Dayfud*, infecting all its inhabitants and its king. Only by working as a team you can defeat the infection and save the castle from its terrible fate. Will a few drops make it to the end?

3.1.3 Game Outline

It is a small-scale 3D multiplayer video game with platformer elements. Each of the three players will control one of the drops (*Red, Blue* or *Green*) and must collaborate to solve the different challenges that each level offers. Each of those levels consists of a medieval environment filled with traps and puzzles that impede the heroes' progress. Combining their wits and unique abilities, *Red, Blue* and *Green* will make their way to the heart of the castle fortress, where a fight with the final boss awaits them: the king of parasites.

3.1.4 Unique Selling Point

- Original puzzles only solvable as a team
- Wide variety of challenges
- Various objects to interact with
- Possibility to play online with friends or strangers
- Possibility of communicating through emotes
- Possibility to customize your character

3.1.5 Similar Competitive Products

The Legend of Zelda: Tri Force Heroes, Among Us, Boomerang Fu

3.2 Narrative

3.2.1 Story

Dayfud was a small medieval town with a big castle surrounded by fortresses. It was populated by small beings who watched over their safety, but one day, the sky open up and something peculiar rained down upon them. They were not conventional drops, they were dark. Soon, the inhabitants of *Dayfud* began to lose control of their bodies and succumb to an evil influence. The king of the castle was not spared from this mysterious rain, and his infection rapidly spread throughout the town.

After several years, the sky opened up once more, but this time the drops that fell were clear and pure, like water. Amongst the falling drops, however, were *Red, Blue* and *Green*, each with mission to stop the infection and restore *Dayfud* to its former glory. To succeed, these droplets should gather their courage and join forces to reach the centre of the castle and free the king from the dark parasite.

3.2.2 Main characters

Players will take control of one of the three colored blobs: *Red, Blue* and *Green*. These characters have a friendly and cheerful appearance. They are diminutive, spherical droplets characterized by their translucent appearance and short limbs.

It should be clarified that although these droplets are the main characters of the story, players will have the ability to choose their preferred color for the character when playing the game.

3.3 Gameplay

3.3.1 Core Mechanics

Players must work together to clear the levels. They must form towers of the necessary height, throw teammates to the appropriate spots (be careful not to throw them into the void!), activate *Buttons*, grab *Special Objects*, etc. All this must be done in the right order, combining the 3 *Items* available at each level. There is a unique solution for each puzzle. Some *Special Objects*, like the *Bomb*, can damage allies, and rushing into the void will penalize. Health is common for all team members, if it reaches zero the game is over.

3.3.2 Actions

Players can perform a total of six actions:

- **Move**: Players can move in any direction (constrained to the X and the Z axes).
- **Grab** players or *Special Objects*: A single player can grab, and pick up, another player right in front of them, forming a tower of two. Also, a single player can grab two-player-tower, forming a tower up to three, but not vice versa. Additionally, they can grab *Special Objects* that allow this feature.
- **Throw** players or *Special Objects*: A single player, located at the bottom of a tower, can throw allies that have been previously picked up, up to one unit forward. It can also throw *Special Objects*, as long as it has grabbed them previously.
- **Use Item**: Each player can make use of the *Item* selected at the beginning of each level. If players are forming towers, only the one at the top will be able to use this ability.
- **Emote**: Players can use the *Emote Wheel* (see Figure 4.9) to select an emote (see Figure 3.4). This is an accessibility feature to provide a means of communication among players. The eight available gestures are: "*Over here*", "*Item*", "*Throw*", "*Pick me up*", "*Celebration*", "*Sorry*", "*Nooo*" and "*OK*".
- **Pause**: Players have the ability to pause the game at any time. When a player pauses the game, it triggers a pause for all players, meaning that the game is temporarily halted for everyone.

3.3.3 Controls

The game has been designed with PC controls in mind, but it is also compatible with gamepad. It supports a variety of controllers, including *PlayStation*, *Xbox*, and *Switch*. In addition, players can rebind keys to suit their preferences.

For better understanding, the default controls are shown below:

Action	**Keyboard**	**Gamepad**
Move	WASD/Arrow Keys	Joystick
Grab	E	B
Throw	Space	A
Use Item	Shift	X
Emote	Tab	Y
Pause	Escape	Start

Table 3.1: Player controls for PC.

3.3.4 Game Entities

The game features various elements that players can interact with, either actively or passively. These include *Interactables*, *Activables*, *Items*, *Special Objects*, and *Collectibles*.

Interactables

These objects possess the unique ability to activate or deactivate *Activables*.

- **Fix Button**: This button can be triggered by players or *Special Objects* when stepped on, and it only has the ability to activate (permanent).
- **Pressure Button**: This button can be triggered by players or *Special Objects* when stepped on, and it has both the ability to activate and deactivate (reversible).
- **Fix Sphere**: This sphere can be triggered only by *Items*, and it only has the ability to activate (permanent).
- **End of Room Area**: This invisible area can be triggered when the three players get inside. It is placed at the end of each room and it only has the ability to activate (permanent). Its main purpose is to serve as a barrier to prevent players from moving back to the previous level and to allow access to the next level.
- **End of Level Pressure Button**: This big button is made of three pressure buttons and it can be triggered when the three players has stepped on each of them. Its main purpose is to announce the completion of the level.

Activables

These objects can be activated or deactivated by *Interactables*.

- **Moving Platform**: This platform can move along any axis, including X, Y, and Z, and it has the capability to transport both players and *Special Objects*.
- **Bridge**: This static platform provides a safe passage for players to move from one side to another without the risk of falling into the void.

Items

These objects grant the player a special ability. Each player can acquire one of these items per game.

- **Bomb generator**: This object spawns a *Bomb* and automatically grabs it.
- **Shooter**: This object spawns bullets that travel in a forward direction.

Special Objects

These objects are instantiated by a specific *Item* and can only exist as a result. One notable feature is their grab-and-throw capability.

- **Bomb**: This object takes three seconds to detonate after being spawned. The explosion reaches an area in one unit radius and it can cause harm to players as well as breaking some structures.

Collectibles

These objects are an additional element of the game that is scattered throughout the levels.

- **Coin**: This object increases the player's score when collected.
- **Life heart**: This object has the ability to restore a player's health when collected.

3.4 Level Design

The game's levels are structured in a recognizable pattern, with each level comprising multiple interconnected sub-levels or rooms. These sub-levels are unlocked sequentially as the players solve the puzzles in each one. To progress to the next room, players must reach the end-of-room area. The difficulty of the puzzles increases in each sub-level and level, creating a scaling challenge for players.

- The first level or **Tutorial** consists of 3 rooms. In this scenario, players will learn the most basic mechanics such as movement and interactions between players and the environment. This will be achieved by presenting the mechanics in a safe environment, or with a trivial solution, so the players can understand their use and combine them with other mechanics in a more complex challenge. This level has no *Item* selection yet.
- The second level or **Level 1** consists of 3 rooms, as the following levels. At this point, object selection will be introduced to the player. The available *Items* in this level are two **Shooters** and one **Bomb Generator**.

The upcoming levels will follow the same structure as *Level 1*, but with different combinations of *Items*. However, the scope of this project stage is limited to the development of the levels described previously.

Below, the complete *Tutorial* level (see Figure 3.1) and the *Level 1* are shown (see Figure 3.2).

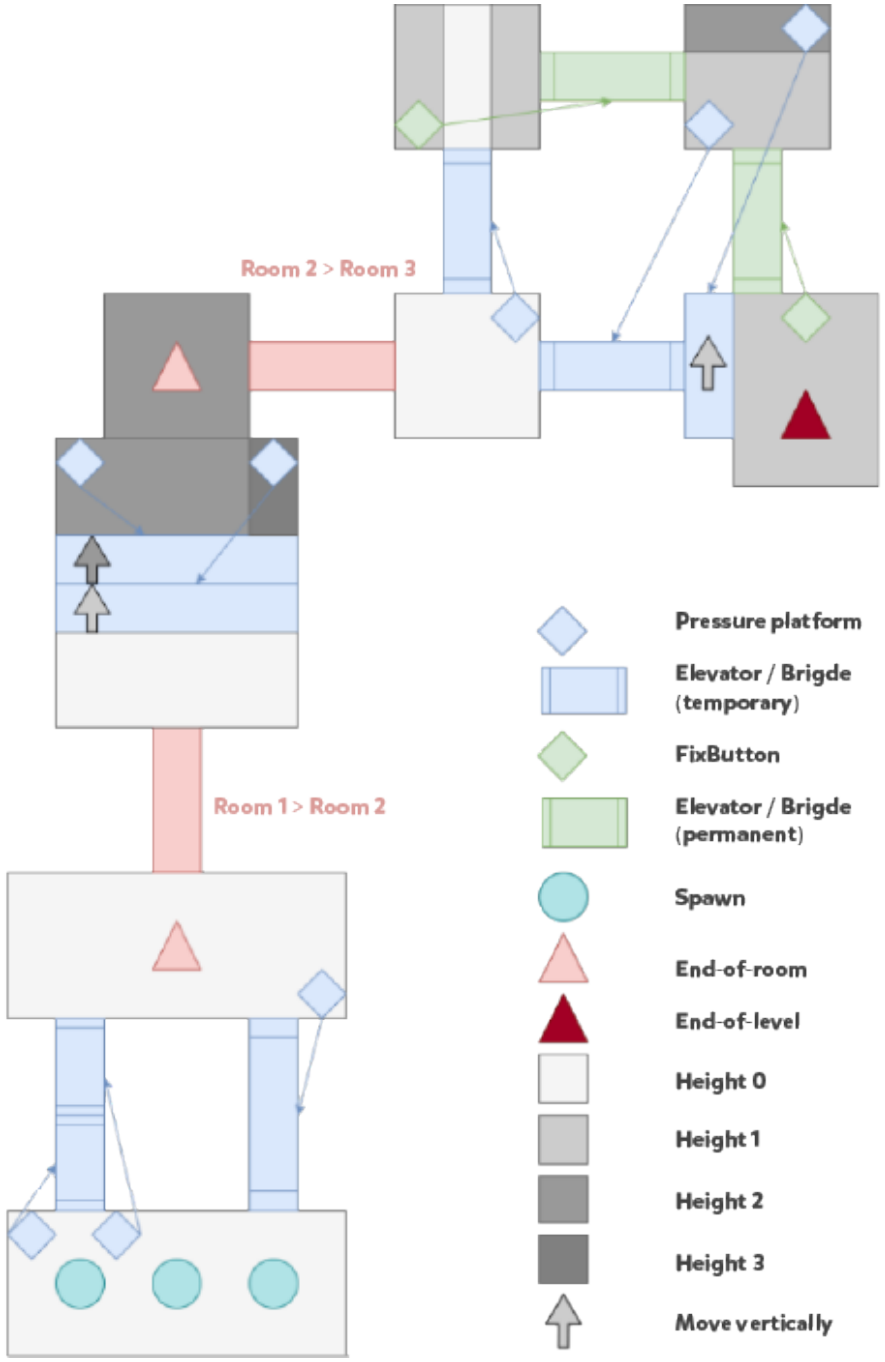

Figure 3.1: Complete Tutorial level preview (made with drawio) [5].

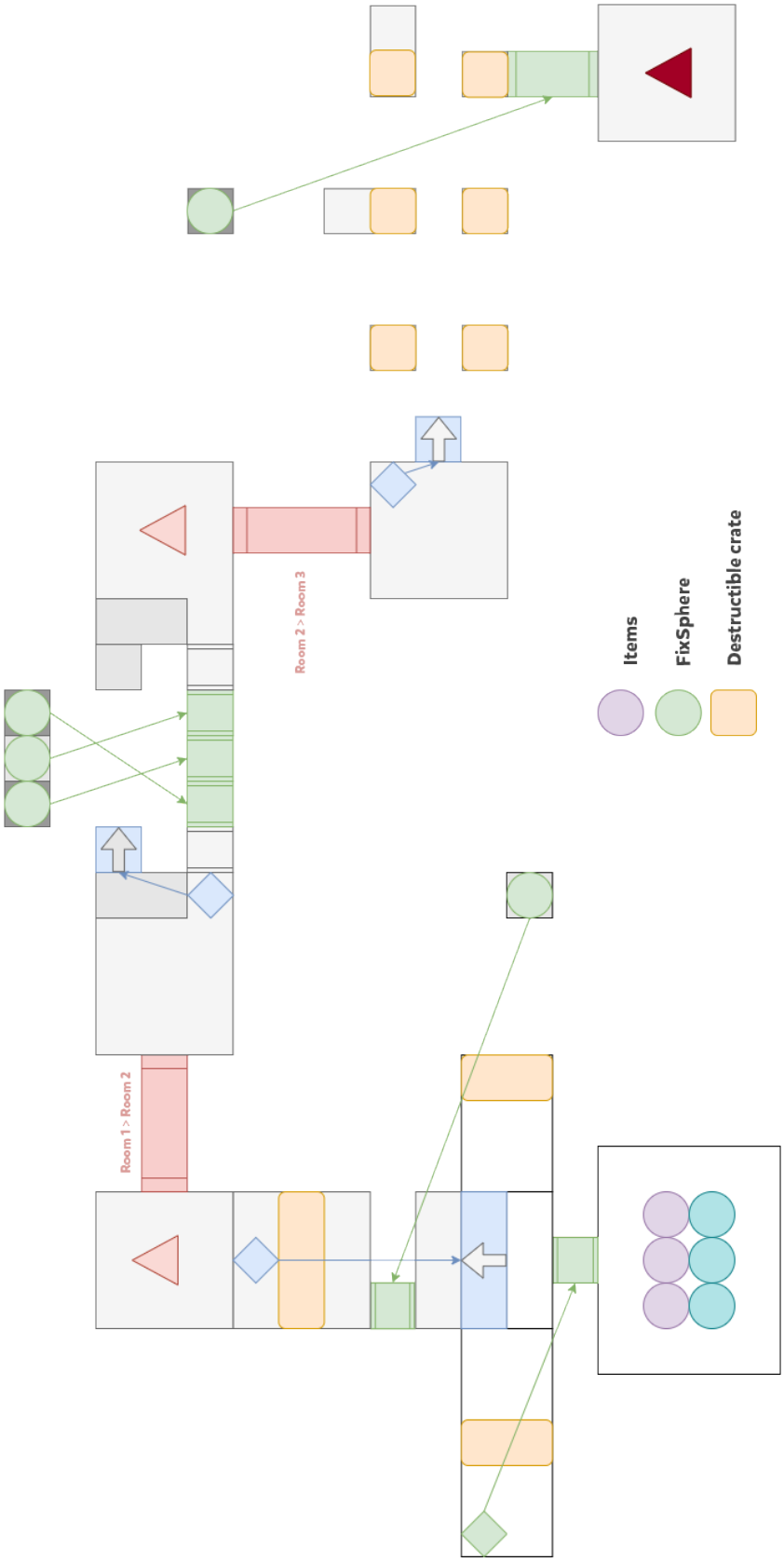

Figure 3.2: Complete Level 1 preview (made with drawio) [5].

3.5 Graphics and Styling

The game is set in a medieval world with floating 3D stylized low poly platforms in a forest-like environment. The camera has a slight tilt, which provides a unique perspective for the game. The artistic direction draws inspiration from a minimalist aesthetic, characterized by a refined color palette consisting of pastel hues, as well as the use of rounded and simplified shapes (see Figures 3.3).

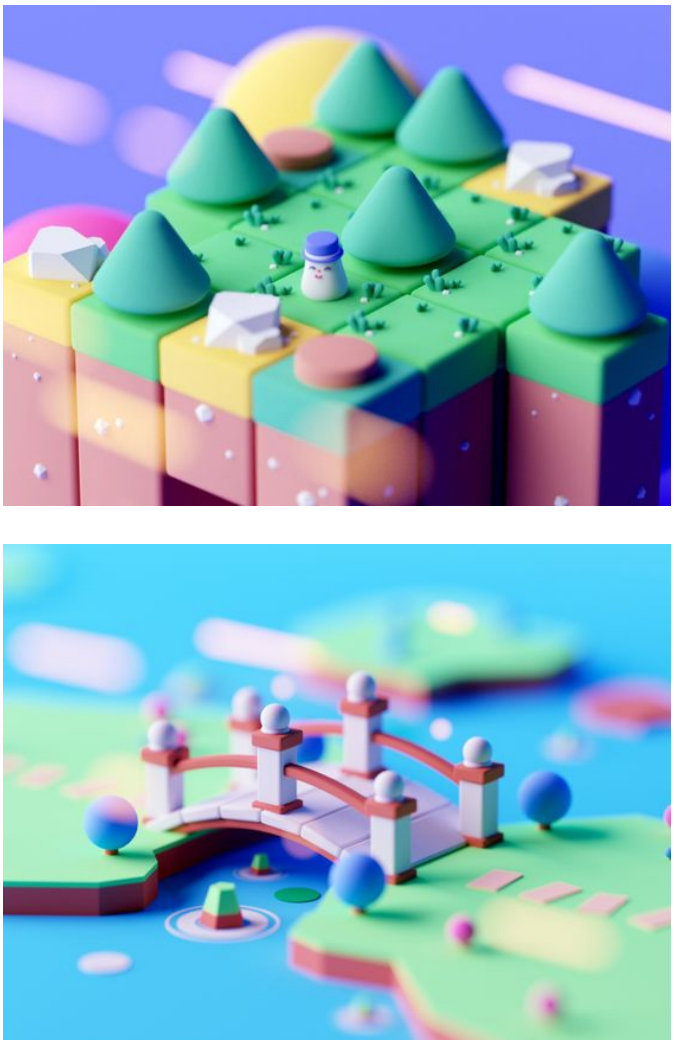

Figure 3.3: Art style references, by *Gustavo Henrique* [6].

3.5.1 Music and Sound

In order to promote player concentration during puzzle-solving levels and facilitate effective communication, the game has deliberately incorporated ambient sound that is both soothing and uplifting. Within the scope of this project, the main track will exclusively play in the menus. However, to enhance immersion, specific player interactions such as walking, throwing, shooting, collecting coins and hearts, bomb explosions, as well as victory and game over screens, will feature unique sound effects. The game's music [17] and sound effects [18] have been sourced from the *Unity Asset Store*.

Figure 3.4: Emote sprite sheet.

CHAPTER 4

System Analysis and Design

Contents

This chapter presents the requirements analysis, design and architecture of the proposed work, as well as an initial exploration of the graphical user interface's usability.

4.1 Requirement Analysis

4.1.1 Functional Requirements

A functional requirement defines a function of the system to be developed, which can be described as a set of inputs, a behavior, and a set of outputs. Such requirements provide a means of defining the system's behavior and enable users to understand its capabilities at any given point in time. Based on this definition, it can be inferred that functional requirements are:

R1. The player can move to the *Lobby* screen by pressing the *Start* button.

R2. The player can adjust the volume of the game and rebind keys by pressing the *Options* button.

R3. The player can see additional information about the game by pressing the *Credits* button.

R4. The player can quit the game by pressing the *Quit* button.

R5. The player can return to the *Main Menu* in any screen of the game by pressing the *Main Menu* button.

R6. The player can move to the *Join Lobby* screen by pressing the *Join* tab.

R7. The player can move to the *Create Lobby* screen by pressing the *Create* Tab.

R8. The player can join a lobby buy pressing the corresponding lobby button on the lobby list.

R9. The player can refresh the lobby list by pressing the *Refresh* button.

R10. The player can join a lobby directly by pressing the *Quick Join* button.

R11. The player can join a lobby with code by typing on the *Code* input field and pressing the *Join* button.

R12. The player can edit their name by typing on the *Name* input field.

R13. The player can create a public lobby by typing on the *Lobby Name* input field and pressing the *Create button.*

R14. The player can create a private lobby by typing on the *Lobby Name* input field, checking the *Private* toggle and pressing the *Create* button.

R15. The player can change their character color by pressing the desired *Color* button.

R16. The player can move to the *Tutorial* screen by pressing the *Ready* button.

R17. The player can start the game by pressing the *Grab* key.

R18. The player can move by pressing *Movement* keys.

R19. The player can grab Players or Special Objects by pressing the *Grab* key.

R20. The player can throw Players or Special Objects by pressing *Throw* key.

R21. The player can use their equipped item by pressing the *Use Item* key.

R22. The player can show an emote by holding the *Emote* key and selecting one.

R23. The player can pause the game by pressing the *Pause* key.

R24. The player can resume the game by pressing the *Resume* button.

R25. The player can select the level by picking one from the *Dropdown* button.

R26. The player can close warning popups by pressing the *Close* button.

R27. The player can collect *Collectibles* such as *Coin* and *LifeHeart*.

R28. The player can interact with *Interactables* such as *FixButton*, *PressureButton*, *EndOfLevel*, *EndOfRoom* and *FixSphere*.

R29. The player can die falling into the void.

4.1.2 Non-functional Requirements

Non-functional requirements are conditions that affect the system's operation, deployment, and maintenance, and cover aspects such as performance, reliability, scalability, security, usability, and maintainability. They need to be carefully considered and addressed throughout the system development life cycle to ensure the system meets the desired quality attributes. The following requirements can be categorized as such:

R30. The player needs access to the Internet connection.

R31. The player needs a minimum of 300 MB of storage for the installation of the game.

R32. The game should be intuitive with a player friendly interface and great visual feedback.

R33. The player preferences, such as the name and the settings, should be saved so they are not lost when the player quits the game.

R34. The player shall be free to join and leave games without penalty.

R35. Errors that may occur during the execution of the game must be notified to the player in a brief and clear manner.

4.2 System Design

In this section, the logical and operational design of the system is presented. The different use cases, which were extracted from the functional requirements, are defined in the following pages along with the **Use case diagram** (see Figure 4.1), **Class diagram** (see Figure 4.2), and **Activity diagram** (see Figure 4.3)

Requirement:	R1
Actor:	Player
Description:	The player can move to the *Lobby* screen by pressing the *Start* button.
Preconditions:	1. The player is at the *Main Menu* screen
Normal sequence:	1. The player presses the *Start* button 2. The scene changes to the *Lobby* screen
Alternative sequence:	None

Table 4.1: Case of use «R1. Lobby Screen»

Requirement:	R2
Actor:	Player
Description:	The player can adjust the volume of the game and rebind keys by pressing the *Options* button.
Preconditions:	1. The player is at the *Main Menu* screen 2. The player is at the *Pause Menu* screen
Normal sequence:	1. The player presses the *Options* button 2. The scene changes to show the *Options* screen
Alternative sequence:	1. The player changes the volume 2. The player rebinds a key

Table 4.2: Case of use «R2. Options Screen»

Requirement:	R3
Actor:	Player
Description:	The player can see additional information about the game by pressing the *Credits* button.
Preconditions:	The player is at the *Main Menu* screen
Normal sequence:	1. The player presses the *Credits* button 2. The scene changes to the *Credits* screen
Alternative sequence:	None

Table 4.3: Case of use «R3. Credits Screen»

Requirement:	R4
Actor:	Player
Description:	The player can quit the game by pressing the *Quit* button.
Preconditions:	The player is at the *Main Menu* screen
Normal sequence:	1. The player presses the *Quit* button 2. The system shutdowns and closes
Alternative sequence:	None

Table 4.4: Case of use «R4. Quit Game»

Requirement:	R5
Actor:	Player
Description:	The player can return to the *Main Menu* in any screen of the game by pressing the *Main Menu* button.
Preconditions:	1. The player is at the *Lobby* screen 2. The player is at the *Character Selection* screen 3. The player is at the *Pause Menu* screen
Normal sequence:	1. The player presses the *Main Menu* button 2. The scene changes to the *Main Menu* screen for the player who performed the action 3. A *Warning Popup* appears on screen notifying the disconnection to the rest of players (see Table 4.26)
Alternative sequence:	None

Table 4.5: Case of use «R5. Main Menu»

Requirement:	R6
Actor:	Player
Description:	The player can move to the *Join Lobby* screen by pressing the *Join* tab.
Preconditions:	The player is at the *Lobby* screen
Normal sequence:	1. The player presses the *Join* tab 2. The scene changes to the *Join Lobby* screen
Alternative sequence:	None

Table 4.6: Case of use «R6. Join Lobby»

Requirement:	R7
Actor:	Player
Description:	The player can move to the *Create Lobby* screen by pressing the *Create* Tab.
Preconditions:	The player is at the *Lobby* screen
Normal sequence:	1. The player presses the *Create* tab 2. The scene changes to the *Create Lobby* screen
Alternative sequence:	None

Table 4.7: Case of use «R7. Create Lobby»

Requirement:	R8
Actor:	Player
Description:	The player can join a public lobby by pressing the corresponding lobby button on the lobby list.
Preconditions:	1. There is at least one public lobby created 2. The lobby is not full
Normal sequence:	1. The player presses the corresponding lobby button 2. The scene changes to the *Character Selection* screen
Alternative sequence:	1. The player presses the corresponding lobby button 2. A network error occurs (see Table 4.26)

Table 4.8: Case of use «R8. Join Public Lobby»

Requirement:	R9
Actor:	Player
Description:	The player can refresh the *Lobby* list by pressing the *Refresh* button.
Preconditions:	The player is at the *Join Lobby* screen
Normal sequence:	1. The player presses the *Refresh* button 2. The system updates the *Lobby* list
Alternative sequence:	None

Table 4.9: Case of use «R9. Refresh Lobby List»

Requirement:	R10
Actor:	Player
Description:	The player can join a lobby directly by pressing the *Quick Join* button.
Preconditions:	The player is at the *Join Lobby* screen
Normal sequence:	1. The player presses the *Quick Join* button 2. The system searches for a public available lobby 3. The system founds an available lobby, the player joins
Alternative sequence:	The system does not found an available lobby, an error occurs (see Table 4.26)

Table 4.10: Case of use «R10. Quick Join»

Requirement:	R11
Actor:	Player
Description:	The player can join a lobby with code by typing on the *Code* input field and pressing the *Join* button.
Preconditions:	1. The player is at the *Join Lobby* screen 2. The player has the corresponding join code
Normal sequence:	1. The player types the code on the *Code* input field 2. The player presses the *Join* button 3. The systems checks if the code is correct 4. The code is correct, the player joins
Alternative sequence:	The code is not correct, an error occurs (see Table 4.26)

Table 4.11: Case of use «R11. Join Lobby Code»

Requirement:	R12
Actor:	Player
Description:	The player can edit their name by typing on the *Name* input field.
Preconditions:	The player is at the *Lobby* screen
Normal sequence:	1. The player presses the *Name* input field 2. The player types a new name
Alternative sequence:	None

Table 4.12: Case of use «R12. Edit Name»

Requirement:	R13
Actor:	Player
Description:	The player can create a public lobby by typing on the *Lobby Name* input field and pressing the *Create button.*
Preconditions:	The player is at the *Create Lobby* screen
Normal sequence:	1. The player types the name on the *Lobby Name* input field 2. The player does not check the *Private* toggle 3. The player presses the *Create* button 4. The systems checks if the name is valid 5. The name is valid, the system creates the lobby and gets the player into the *Character Selection* screen
Alternative sequence:	The name is not valid, the system does nothing.

Table 4.13: Case of use «R13. Create Public Lobby»

Requirement:	R14
Actor:	Player
Description:	The player can create a private lobby by typing on the *Lobby Name* input field, checking the *Private* toggle and pressing the *Create* button.
Preconditions:	The player is at the *Create Lobby* screen
Normal sequence:	1. The player types the name on the *Lobby Name* input field 2. The player checks the *Private* toggle 3. The player presses the *Create* button 4. The systems checks if the name is valid 5. The name is valid, the system creates the lobby and gets the player into the *Character Selection* screen
Alternative sequence:	The name is not valid, the system does nothing.

Table 4.14: Case of use «R14. Create Private Lobby»

Requirement:	R15
Actor:	Player
Description:	The player can change their character color by pressing the desired *Color* button.
Preconditions:	1. The player is at the *Character Selection* screen 2. The desired color is available
Normal sequence:	1. The player presses the desired *Color* button 2. The system changes the player's character color
Alternative sequence:	None

Table 4.15: Case of use «R15. Character Color»

Requirement:	R16
Actor:	Player
Description:	The player can move to the *Tutorial* screen by pressing the *Ready* button.
Preconditions:	The player is at the *Character Selection* screen
Normal sequence:	1. The player presses the desired *Color* button 2. The system checks if all the players in the lobby are ready 3. All the players are ready, the scene changes to the *Tutorial* screen
Alternative sequence:	Not all the players are ready, the system does nothing

Table 4.16: Case of use «R16. Player Ready»

Requirement:	R17
Actor:	Player
Description:	The player can start the game by pressing the *Grab* key.
Preconditions:	The player is at the *Tutorial* screen
Normal sequence:	1. The player presses the *Grab* key 2. The system checks if all the players in the *Tutorial* screen have pressed the *Grab* key 3. All the players have pressed the *Grab* key, the scene changes to the *Game* screen
Alternative sequence:	Not all the players have pressed the *Grab* key, the system does nothing

Table 4.17: Case of use «R17. Game Start»

Requirement:	R18
Actor:	Player
Description:	The player can move by pressing *Movement* keys.
Preconditions:	1. The player is at the *Game* screen 2. The player's state must be BOT
Normal sequence:	1. The player presses the *Movement* keys 2. The player moves
Alternative sequence:	None

Table 4.18: Case of use «R18. Movement»

Requirement:	R19
Actor:	Player
Description:	The player can grab by pressing the *Grab* key.
Preconditions:	1. The player is at the *Game* screen 2. The player's state must be BOT 3. The player has something to grab in range
Normal sequence:	1. The player presses the *Grab* key 2. The player grabs
Alternative sequence:	None

Table 4.19: Case of use «R19. Grab»

Requirement:	R20
Actor:	Player
Description:	The player can throw by pressing *Throw* key.
Preconditions:	1. The player is at the *Game* screen 2. The player's state must be BOT 3. The player is holding something
Normal sequence:	1. The player presses the *Grab* key 2. The player throws what is holding
Alternative sequence:	None

Table 4.20: Case of use «R20. Throw»

Requirement:	R21
Actor:	Player
Description:	The player can use their equipped item by pressing the *Use Item* key.
Preconditions:	1. The player is at the *Game* screen 2. The player's state must be BOT or TOP 3. The player has equipped an item
Normal sequence:	1. The player presses the *Use Item* key 2. The player uses their item
Alternative sequence:	None

Table 4.21: Case of use «R21. Use Item»

Requirement:	R22
Actor:	Player
Description:	The player can show an emote by holding the *Emote* key and selecting one.
Preconditions:	The player is at the *Game* screen
Normal sequence:	1. The player presses the *Emote* key 2. The player selects an emote 3. The player displays the selected emote on the screen
Alternative sequence:	1. The player presses the *Emote* key 2. The player does not select an emote 3. The player displays the last selected emote, or default, on the screen

Table 4.22: Case of use «R22. Emote»

Requirement:	R23
Actor:	Player
Description:	The player can pause the game by pressing the *Pause* key.
Preconditions:	The player is at the *Game* screen
Normal sequence:	1. The player presses the *Pause* key 2. The system pauses the *Game* screen 3. The system actives the *Pause* screen 4. The system notifies the other players that the game has been paused
Alternative sequence:	None

Table 4.23: Case of use «R23. Pause»

Requirement:	R24
Actor:	Player
Description:	The player can resume the game by pressing the *Resume* button.
Preconditions:	The player is at the *Pause* screen
Normal sequence:	1. The player presses the *Resume* button 2. The system unpauses the *Game* screen 3. The system disables the *Pause* screen
Alternative sequence:	1. The player presses the *Pause* key 2. The system unpauses the *Game* screen 3. The system disables the *Pause* screen

Table 4.24: Case of use «R24. Resume»

Requirement:	R25
Actor:	Player
Description:	The player can select the level by picking one from the *Dropdown* button.
Preconditions:	The player is at the *Create* tab at the *Lobby* screen
Normal sequence:	1. The player presses the *Dropdown* button 2. The player selects a level
Alternative sequence:	None

Table 4.25: Case of use «R25. Level Selector»

Requirement:	R26
Actor:	Player
Description:	The player can close *Warning Popups* by pressing the *Close* button.
Preconditions:	1. An error has occurred during the *Game* scene 2. An error has occurred during the *Lobby* scene
Normal sequence:	1. The *Warning Popups* appears during the *Game* scene 2. The player presses the *Close* button 3. The player returns to the *Main Menu* screen
Alternative sequence:	1. The *Warning Popups* appears during the *Lobby* scene 2. The player presses the *Close* button 3. The *Warning Popup* closes

Table 4.26: Case of use «R26. Warning Popup»

Requirement:	R27
Actor:	Player
Description:	The player can collect *Collectibles* such as *Coin* and *LifeHeart*.
Preconditions:	The player is at the *Game* screen
Normal sequence:	1. The player collides with a *Coin* 2. The player earns a coin
Alternative sequence:	1. The player collides with a *LifeHeart* 2. The player gains health

Table 4.27: Case of use «R27. Collect»

Requirement:	R28
Actor:	Player
Description:	The player can interact with *Interactables* such as *FixButton*, *PressureButton*, *EndOfLevel*, *EndOfRoom* and *FixSphere*.
Preconditions:	The player is at the *Game* screen
Normal sequence:	1. The player collides with a *Interactable* 2. The *Interactable* gets active
Alternative sequence:	None

Table 4.28: Case of use «R28. Interact»

Requirement:	R29
Actor:	Player
Description:	The player can die falling into the void.
Preconditions:	1. The player is at the *Game* screen 2. The player has fell into the void
Normal sequence:	1. The system removes a life from the players and checks new players' health 2. Players health is greater than zero, the player respawns in the last safe position.
Alternative sequence:	Players health is not greater than zero, The scene changes to the *Game Over* screen

Table 4.29: Case of use «R29. Die»

4.2.1 Use Case diagram

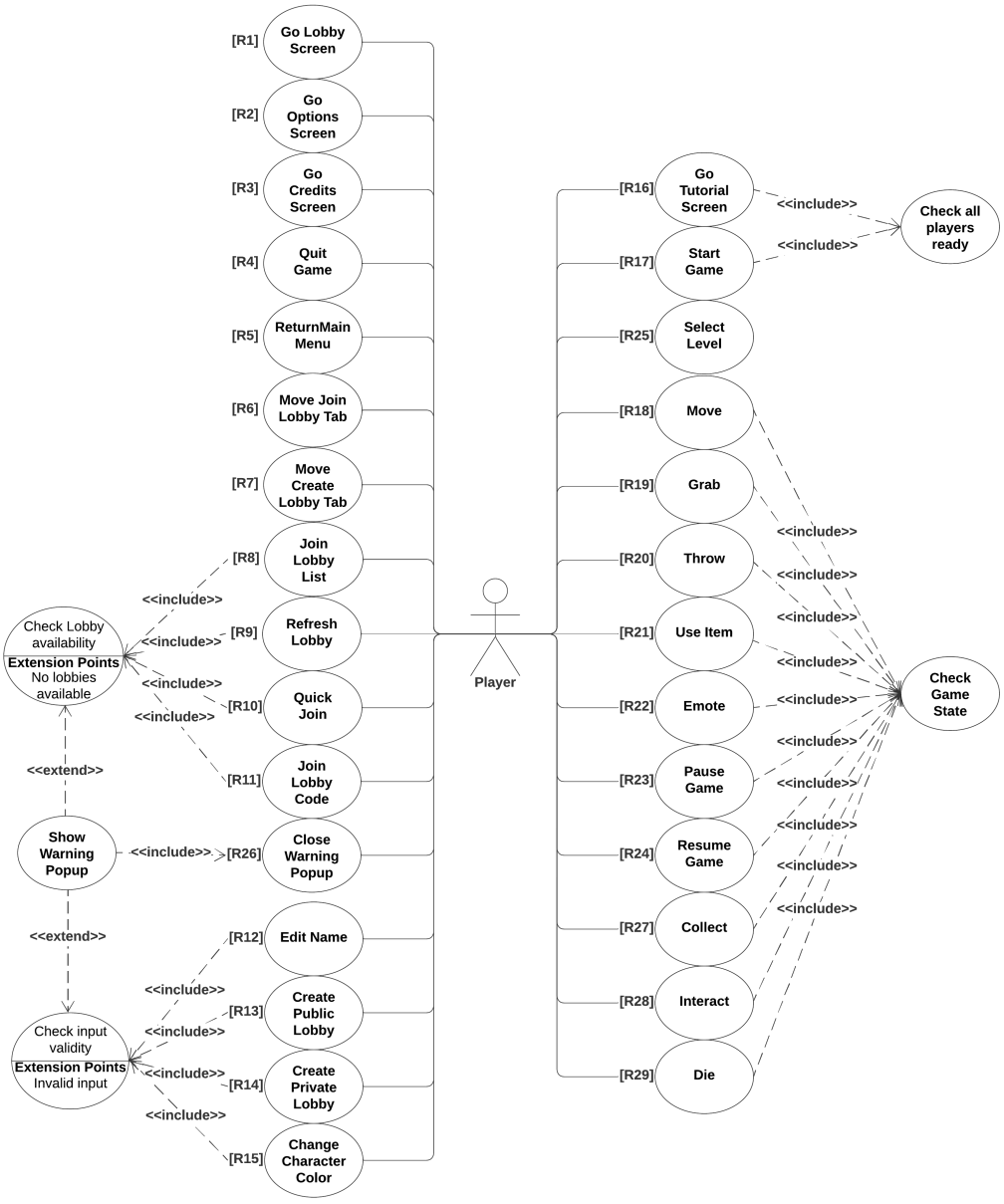

Figure 4.1: Case of use Diagram (made with Lucidchart) [10].

4.2.2 Class diagram

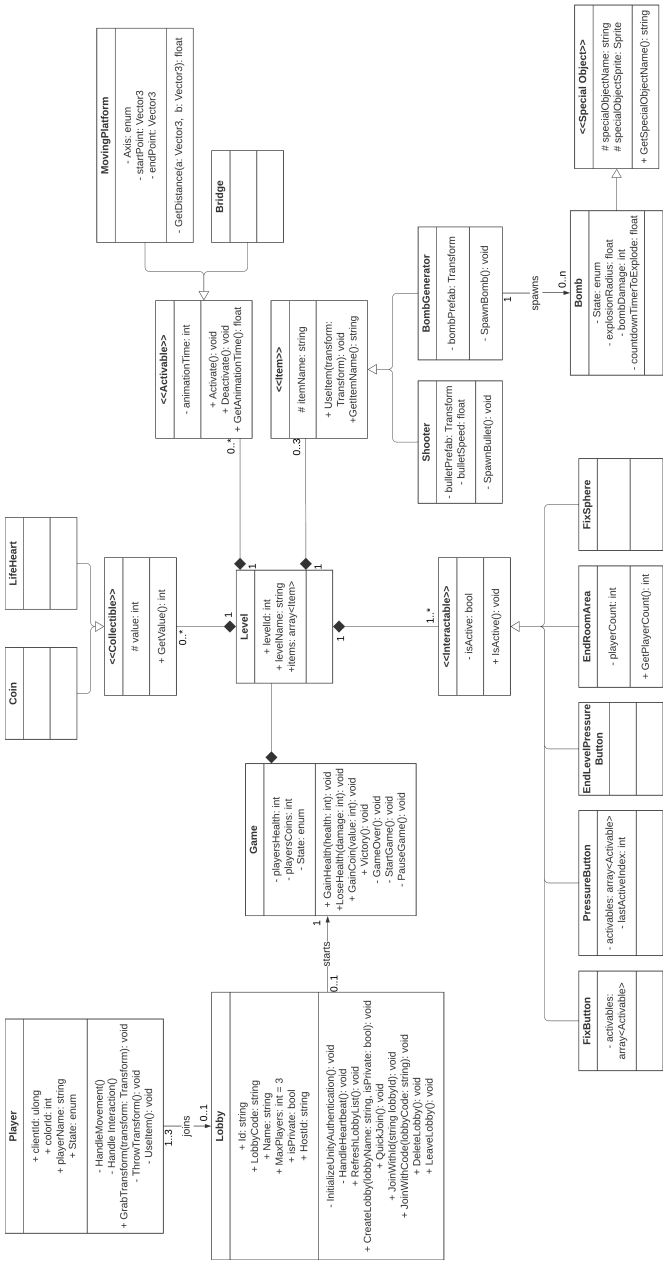

Figure 4.2: Class Diagram (made with Lucidchart) [10].

4.2.3 Activity diagram

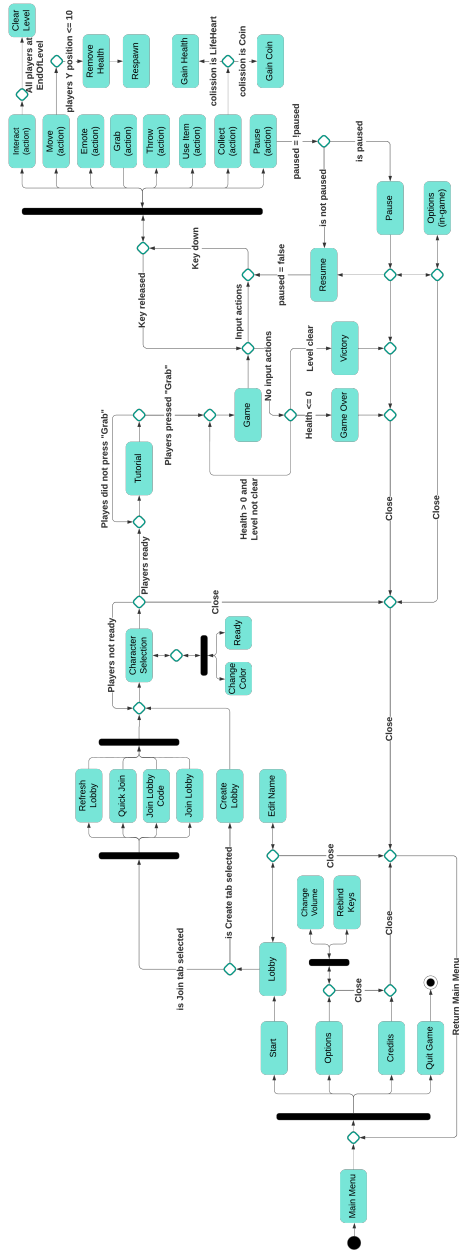

Figure 4.3: Activity Diagram (made with Lucidchart) [10].

4.3 System Architecture

The system requirements of a computer play a crucial role in determining whether it can run specific software or games. Due to the limited number of computers on which the game has been tested, its functionality cannot be guaranteed under circumstances other than the following suggested:

- **Operating System**: Windows 10 64-bit
- **CPU**: Intel Core i7 7th Gen
- **GPU**: Nvidia GeForce GTX 1050 Ti
- **RAM**: 8 GB

Therefore, in orther to run this game effectively, the following system requirements are suggested by the official Unity's documentation [23]:

- **Operating System**: Windows 7 (SP1+), Windows 10 and Windows 11
- **CPU**: x86, x64 architecture with SSE2 instruction set support
- **GPU**: DX10, DX11, DX12 capable

In addition to meeting the hardware requirements, it is mandatory to have access to a stable Internet connection to enjoy the full features of the game. The online multiplayer mode relies heavily on the internet to ensure a seamless and uninterrupted gameplay experience.

4.4 Interface Design

The game's User Interface (UI) features a minimalist design with a pastel color scheme. Only essential information is displayed, presented in a way that is easily recognizable to the user. To display text on the screen, the font *Mikado Bold* will be used (see Figure 4.4).

Figure 4.4: *Joint Drops* game title typed using *Mikado Bold*.

The game will have nine screens in total:

- The **Main Menu**: This is the main screen of the game. It displays the game's title and provides access to several options, including the Start, Options, Credits, and Quit buttons (see Figure 4.5).

Figure 4.5: Main Menu screen with the game Splash Art (made with Krita) [8].

- The **Lobby**: This screen allows players to join or create a game session, as well as edit their player name. For creating or joining a lobby, the payer can switch between the following tabs:

  - **Join** tab: The default tab when loading the *Lobby* scene, which provides options for players to *Join* a lobby from the available list, *Join by Code*, or use the *Quick Join* feature. Additionally, it allows players to refresh the lobby list as well as edit their player name (see Figure 4.6).

  - **Create** tab: This tab enables players to create a lobby by entering a name, choosing the desired level to play, and selecting whether the lobby should be public or private (see Figure 4.7).

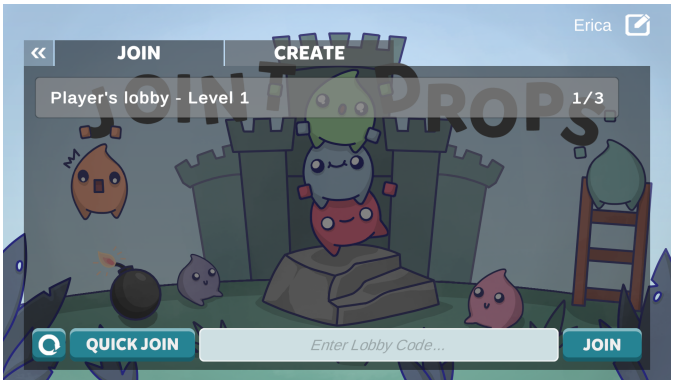

Figure 4.6: Join tab.

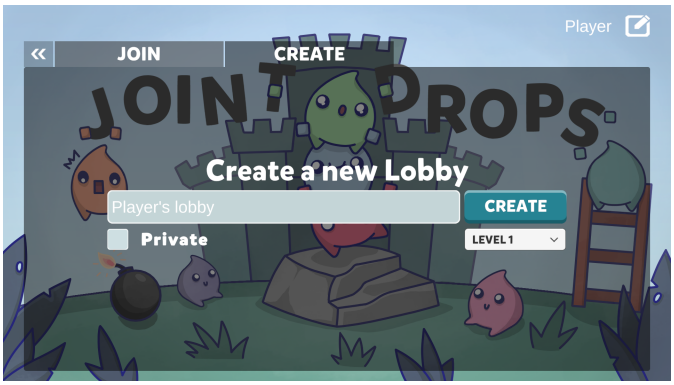

Figure 4.7: Create tab.

- The **Character Selection**: This screen allows players to customize their character color for the game, providing a range of options to choose from (see Figure 4.10).

- The **Game**: This is the active screen when the game is running (see Figure 4.9). It has a Head-Up Display (**HUD**) showing the players' health, the level name and the coins obtained (see Figure 4.8).

Figure 4.8: UI elements (made with Krita) [8].

Figure 4.9: Game screen showing the emote wheel.

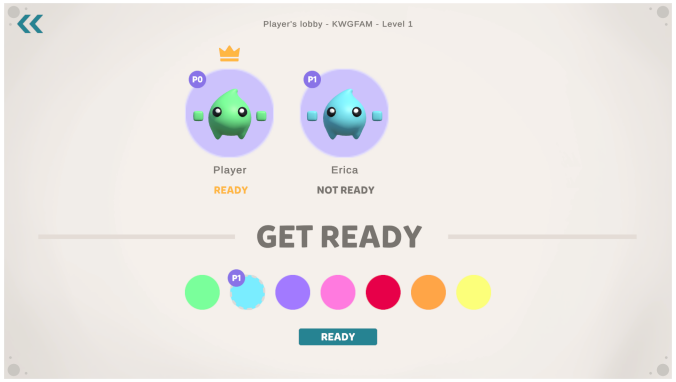

Figure 4.10: Character Selection screen.

- **Victory**: This screen is displayed when players successfully complete the game. It showcases the final stats, including the number of collected coins. Additionally, players are provided with a button to play again. Upon clicking the button, the lobby is dissolved, and players are redirected to the *Main menu*.

- **Game Over**: This screen is displayed when players fail to complete the level and lose the game. It showcases the final stats, including the number of collected coins. Additionally, players are provided with a button to play again. Upon clicking the button, the lobby is dissolved, and players are redirected to the *Main menu*.

- The **Pause**: This screen can be accessed during gameplay when a player pauses the game. It displays several buttons, including *Resume*, *Options*, and return to *Main Menu*. Whenever a player initiates a pause in the game (see Figure 4.11), it applies a pause to all players, triggering a notification for everyone involved (see Figure 4.12).

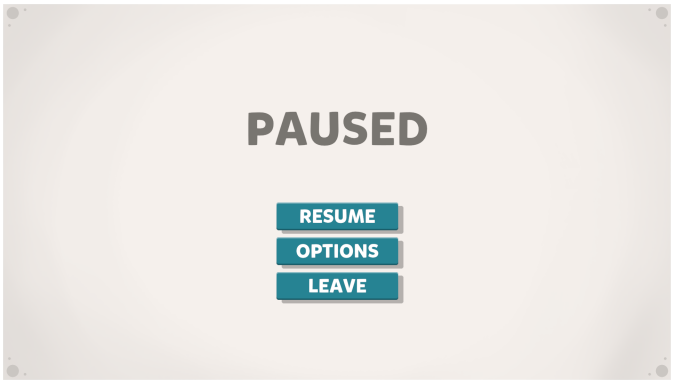

Figure 4.11: Pause screen for the player who pauses the game.

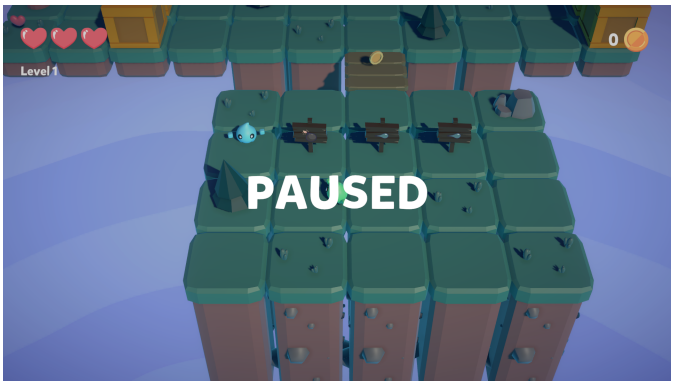

Figure 4.12: Pause screen for the rest of players.

- **Credits**: This screen displays information about the developer of Joint Drops.
- **Options**: This screen provides access to the game's control and audio settings, including the ability to rebind keys to different actions (see Figure 4.13).

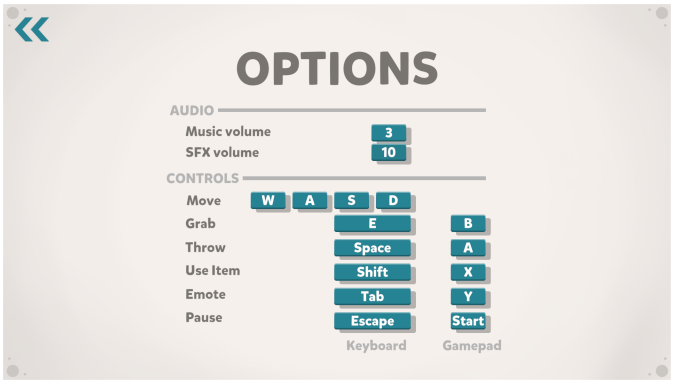

Figure 4.13: Options screen.

Chapter 5

Work Development and Results

Contents

This chapter is a detailed review of all the work carried out, with special emphasis on the procedures followed to obtain the results and justifying the ways of proceeding.

5.1 Work Development

To enhance readability and comprehension, this section has been divided into its core sections: Programming, Multiplayer and Game Art. These follow a chronological order, according to the planning and the importance of the achieved milestones in each of them.

5.1.1 Programming

There are different approaches to consider when planning the development of a multiplayer game. One approach is to implement the entire game offline and then adapt it to multiplayer, while the other is to develop both aspects concurrently.

In this particular project, it was chosen to implement the game entirely offline first, given the lack of experience in implementing a multiplayer system and developing long-term projects. This decision proved successful, but the experience gained from this project has given a broader perspective on game development needs, so the option of adopting the other approach in future projects is not disregarded.

With a clear approach in mind, the development began by implementing the primary mechanics of the game. Each of these will be detailed below, following the order of implementation:

1. **Player actions and gamepad feature**: Unity's built-in input system have been utilized to detect and handle different inputs from the player, such as keyboard and mouse or gamepad controls for the player actions. Animations and physics have also been implemented to make player movements and interactions with the environment more realistic. The last ones have been implemented independently, as there was no requirement to merge functionality with visuals for the sake of programming neatness.

   It is worth mentioning that the throwing action has been implemented by drawing a parabolic trajectory using mathematical calculations instead of physics-based calculations, which allows for better control over the curvature without relying on forces or gravity (see Appendix A). This method was inspired by a developer thread [14]. However, this approach has led to some challenges as multiple collision checks need to be performed to prevent any inaccuracies in the results.

   Additionally, to prevent collisions, players spawn at different locations at the start of each game. Furthermore, user preferences such as name, sound, and key settings are saved using *Unity PlayerPrefs* for a personalized gaming experience [4].

2. **Emote wheel**: The emote wheel have been programmed using UI to display a wheel with different emotes for the player to select (see Figure 4.9). This emotes are made with *Scriptable Objects* (a Unity data class) for saving information such as the id, the name, and the icon image. The size of the wheel is automatically adjusted based on the number of emotes present, making it easy to incorporate modifications.

3. **Collectibles, Interactables, Activables**: These elements serve as the parent class that all derived classes can inherit from in order to utilize shared functionalities. Each of them include logic for detecting collisions, handling animations, and updating game states based on the player's interactions with these objects.

4. **Items, Special Objects**: Like the previous elements, these two also act as the parent class, providing a base for other classes to inherit and utilize common features. The *Items* in the game serve as a means of informing the player about the *Special Object* or ability they have acquired. Each *Special Object*, however, will exhibit unique behavior when used.

   Currently, the game features only one *Special Object*, the *Bomb*, which is equipped with a state machine to control its behavior. The state machine comprises three distinct states, *Idle, ExplosionCountdown, and Exploded*, which are responsible for controlling the bomb's actions, including detonation and self-destruction after a 3-second countdown.

5. **Game Screens and Head-Up Display (HUD)**: The different game screens have been implemented using Unity's Scene Manager and UI elements. Also the HUD has been programmed to display relevant information to the player during gameplay like health, coins and the level name.

   Moreover, to conceal any delays when transitioning between screens, a *Loading* screen has also been implemented. Its purpose is to provide a seamless experience during screen switches. Furthermore, to ensure consistent functionality across all buttons in the game, a dedicated script has been developed to manage the *OnPointerEnter, OnPointerExit, OnPointerDown*, and *OnPointerUp* events for seamless interaction with the buttons.

6. **Game Manager, Health system, Sound and Music system**: The Game Manager have been implemented to handle different game states, such as *WaitingToStart, CountdownToStart, GamePlaying, Victory and GameOver*. Additionaly, a health system has been programmed to manage the player's health during gameplay. This system allows the player to take damage as well as respawning.

   Finally, the Sound and Music system has been autonomously integrated using event triggers, enabling centralized control from a single script. The system leverages a separate *Scriptable Object* to store and manage all audio clips utilized in the game.

5.1.2 Multiplayer

Before diving into the multiplayer functionality, it was crucial to conduct a thorough study of its operation and determine the appropriate mode of authority for the network. The two modes of authority are server-authority and client-authority [21].

- In **server-authority**, the game server controls the game logic, validating the actions of the clients and sending the updates to all the players. This mode provides a more secure gameplay experience as the server can validate every action, preventing hacking and cheating. However, it can also result in slower gameplay due to the constant communication between the server and clients.
- In **client-authority**, the game logic is controlled by the clients, and the server validates the actions of the players and corrects any errors. This mode can result in faster gameplay as there is less communication among the server and clients. However, it can be more vulnerable to hacking and cheating as the clients have more control over the game.

Nevertheless, after careful consideration and analysis, it was decided to implement the client-authority mode for the multiplayer functionality of the game. This decision was based on the game's low-scale cooperative nature, designed to be played with friends, which made it less vulnerable to hacking concerns typically found in competitive games. Prioritizing responsiveness over reliability on a single instance was deemed more appropriate for the game's intended audience and gameplay.

The following consideration was whether to allow late joins in the game, which would enable players to join a match that had already begun. However, given that the game is specifically designed for three players, it would not make sense to start with fewer players or to continue if one player left. Therefore, it was decided that late joins would not be permitted in this instance.

Unity Netcode offers two options for synchronization in multiplayer: Network Variables and Remote Procedure Calls, making the operation of multiplayer relatively straightforward.

- **Network Variables** are variables that can be synchronized across the network. When a network variable is changed on a client, it will automatically be updated on all other clients in the game. These have been use for tracking player health, game states and coins.
- **Remote Procedure Calls (RPCs)** are functions that can be called across the network. When an RPC is called on a client, it will be executed on all other clients in the game. This has been used for implementing game mechanics such as spawning a projectile or playing an animation.

Both options, Network Variables and RPCs, can be used for synchronization in Unity Netcode since it was decided not to allow late joins, which is the only drawback of RPCs.

To ensure synchronization among players, a *Network Manager* was added to display all objects existing in the network context. All these objects required the *Network Object* component to be attached. In order for the players to move around, they needed a *Network Transform* component, which is not provided in the inspector due to the default server-authority mode. Instead, Unity provides the necessary code in the documentation.

To synchronize player movement, it was utilized the *ClientNetworkTransform* [21] component from the documentation, as well as the *OwnerNetworkAnimator* component [20] to synchronize animations (see Appendix A).

To synchronize the remaining actions and objects of the game, it has been utilized RPCs, following the same approach. It is important to note that the network architecture being utilized is based on the client-server, where the host acts as both the client and server, with the other connections acting solely as clients. In *Netcode for Gameobjects*, the server has the authority to perform game actions such as spawning and destroying objects, editing variables, reparenting and more. As a result, any client who wants to perform such actions must request the server's permission. The request structure is as follows:

1. The client detects the need for an action and calls the *ServerRpc* function, which can only be executed by the server (without ownership).

2. The server receives the petition and calls the *ClientRpc* function, which can be executed by all clients, where the code will be executed.

In this manner, the server is responsible for completing the task and ensuring that the outcome is disseminated to all clients (see Appendix A).

Despite the conveniences provided by the tool, it is important to note that it still contains bugs, as it is a newly developed tool that is still undergoing refinement. Some of these malfunctions are related to parenting. In the presented game, one of the key mechanics is the ability to grab players and *Special Objects*. Initially, this was implemented by making the grabbed object a child of the grabber player in the hierarchy, which resulted in sharing global coordinates with the movement. However, this method proved to be unsuitable for synchronization in the multiplayer mode. As a result, a change in the approach was necessary. A new component called *FollowTransform* (see Appendix A) has been created instead of reparenting, which allows the player or object being grabbed to copy the coordinates of its parent, resulting in better outcomes and better adaptation to multiplayer.

The next objective was to achieve synchronization of game states, pauses, and disconnections. Additionally, a *Lobby* system needed to be implemented that could meet all the requirements of the game, including:

- A display of available public lobbies updated in real-time. So the player can choose from a list.
- An option to quickly join the first game that is available.
- An option to join a game using a code.
- An option to create games with the ability to select between public or private.

Lastly, the *Relay* was implemented, a service that allows players from anywhere in the world to connect through a server. This task was relatively straightforward compared to all the previous ones since it only required a minor modification to the *Lobby* logic to make it utilize an *Allocation* (see Appendix A).

5.1.3 Game Art

During the asset modeling phase of the game, a cute, low-poly minimalist style was employed, with simple shapes and easily recognizable elements. The main player character, designed to resemble a small water drop, was created along with other objects and all game world's elements, using the same style (see Figure 5.1).

To ensure smooth, realistic movements that complement the game's style, the player character's animations were rigged, including *idle, idle grabbing, walking, walking grabbing*, and *throwing*. To facilitate future customization options, the character's texturing has been intentionally kept minimal, as players can modify the character's color later on in the game (see Figure 4.10). The collectibles, including the coin and the heart, have also been animated.

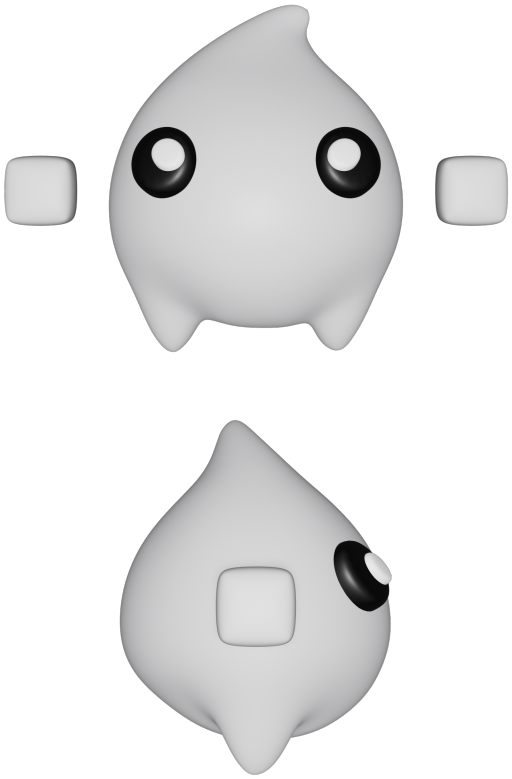

Figure 5.1: Main character modeling (made with Blender) [2].

Post-processing effects like vignette and color alteration were added to improve the game's visuals, making it more refined and polished. Moreover, a *cartoon shader* was used to give the player a more distinct and unique appearance, making them stand out from other game elements.

Additionally, an eye-catching splash art illustration was created to represent the game's style and setting, attracting potential players and providing them with a sense of what to expect (see Figure 4.5). Furthermore, the Unity's particle system was utilized to create visual effects, such as the *Bomb* explosion and the destructible crate effect and confetti in the *Credits* screen. The *ProBuilder* tool was also used to create and edit object geometry that breaks into pieces, adding another layer of immersion to the game.

The implementation of levels involved designing the layout of each level, with the necessary obstacles, collectibles, and interactive objects in place. Every level required appropriate decoration and scenery to suit the game's setting. The process involved striking a balance between gameplay mechanics and aesthetics to ensure that the levels were not only fun to play but also visually appealing and engaging (see Figure 5.2). Lastly, it is important to note that the skybox asset utilized in the game to enhance the visual atmosphere, have been sourced from the *Unity Asset Store* [16].

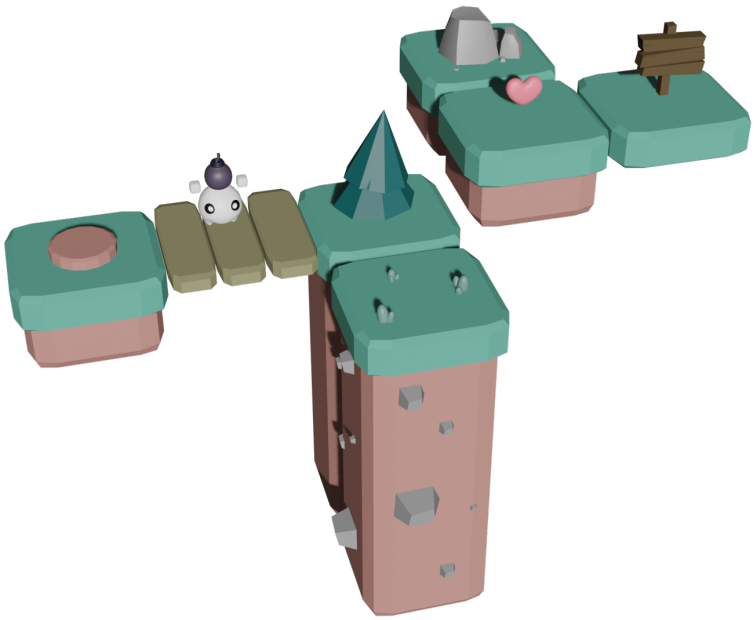

Figure 5.2: Game world's elements.

5.2 Results

Ultimately, despite facing some challenges along the way, the project's initial planning was able to flexibly adapt to the evolving requirements and successfully achieve all six of the original objectives.

The game has been successfully developed in Unity with a strong programming foundation that allows for future expansion and development of new game mechanics. All of the main mechanics that were designed, such as player actions, map elements, items, and early levels, have been incorporated. The multiplayer system has also been implemented effectively, enabling players to play online with others from around the world, create and join lobbies. Additionally, the game has been given its unique artistic style by creating assets and other visual elements.

This project can serve as a blueprint for developing other multiplayer games, as it lays out the essential foundations needed for such games. Developers can build upon the existing network architecture, synchronization methods, and lobby system, adapting it to fit their specific needs and requirements.

Furthermore, the lessons learned during the development of this project can be used to agile the development of future multiplayer games. The challenges and solutions encountered during the creation of this game can be applied to other projects, avoiding common pitfalls and optimizing the development process.

The work results can be checked by playing the game or inspecting files available in the following links:

- **Github repository**: https://github.com/Slinon/Joint-Drops
- **Google Drive repository**: https://drive.google.com
- **Official website**: https://slinon.itch.io/joint-drops

CHAPTER 6

Conclusions and Future Work

Contents

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

6.1 Conclusions

In conclusion, the learning experience garnered from this project has been immense. It has enabled me to cultivate a knack for organizing larger and more complex projects, honing my programming skills to create cleaner and more optimized code, and also equipped me with the ability to develop a seamless multiplayer system in Unity.

However, developing a multiplayer system can be a daunting task, as it requires a significant amount of time and resources. When incorporating a multiplayer feature, the duration of the project is likely to double the estimated time. While creating a basic multiplayer system may be a feasible task, producing a refined, anti-cheat, accurate, and fair multiplayer system requires a considerable amount of time and effort.

Also, encountering bugs in Unity and having to start over to achieve the expected behavior could be frustrating at times.

'I never once failed at making a light bulb.
I just found out 99 ways not to make one.'
- Thomas A. Edison

Despite of that, this experience has taught me valuable problem-solving skills that I can apply to future projects, speeding up development and reducing time costs.

6.2 Future work

The nature of the game encourages scalability, so it is planned to resume the project in the near future by adding more levels, objects, and improving the game mechanics and artistic design. It is even possible to consider a marketing campaign and publishing the game on platforms such as Steam [15].

As the project expands, it may require a larger team to ensure a quality product and timely delivery. Therefore, it may be necessary to contact a game designer, musical composer, 3D and 2D artist, and additional programmers for the project's expansion.

Maintaining and improving the servers of successful multiplayer video games is an ongoing process. As the service grows, it may become unsustainable for a single person to handle.

Bibliography

[1] Atlassian. Trello. https://trello.com/es. Accessed: 2023-05-04.

[2] Blender. Blender 3.5. https://www.blender.org/. Accessed: 2023-05-04.

[3] Canva. Canva. https://www.canva.com/. Accessed: 2023-05-04.

[4] Unity Documentation. Playerprefs. https://docs.unity3d.com/ScriptReference/PlayerPrefs.html. Accessed: 2023-05-19.

[5] drawio. diagrams.net. https://www.drawio.com/. Accessed: 2023-05-18.

[6] Gustavo Henrique. Dribbble. https://dribbble.com/guutv. Accessed: 2023-05-22.

[7] Jobted. Sueldo del programador de videojuegos en españa. https://www.jobted.es/salario/programador-videojuegos. Accessed: 2023-05-04.

[8] Krita. Digital painting. https://krita.org/es/. Accessed: 2023-05-04.

[9] Liquidsky. Why are multiplayer games more popular than single player games? https://liquidsky.com/why-are-multiplayer-games-more-popular-than-single-player-games/. Accessed: 2023-05-04.

[10] Lucid. Lucidchart. https://www.lucidchart.com/. Accessed: 2023-05-19.

[11] Microsoft. Github desktop. https://desktop.github.com/. Accessed: 2023-05-04.

[12] Microsoft. Visual studio community 2022. https://visualstudio.microsoft.com/vs/community/. Accessed: 2023-05-04.

[13] Overleaf. Overleaf. https://www.overleaf.com/. Accessed: 2023-05-04.

[14] GameDev StackExchange. Add parabola curve to straight movetowards() movement. https://gamedev.stackexchange.com/questions/183507/add-parabola-curve-to-straight-movetowards-movement. Accessed: 2023-05-18.

[15] Steam. Steam store. https://store.steampowered.com/. Accessed: 2023-05-05.

[16] BOXOPHOBIC Unity Asset Store. Polyverse skies | low poly skybox shaders. https://assetstore.unity.com/packages/vfx/shaders/polyverse-skies-low-poly-skybox-shaders-104017. Accessed: 2023-05-19.

[17] Rizwan Ashraf Unity Asset Store. Free music tracks for games. https://assetstore.unity.com/packages/audio/music/free-music-tracks-for-games-156413. Accessed: 2023-05-18.

[18] SwishSwoosh Unity Asset Store. Free ui click sound pack. https://assetstore.unity.com/packages/audio/sound-fx/free-ui-click-sound-pack-244644. Accessed: 2023-05-18.

[19] Unity. Netcode for gameobjects. https://docs.unity3d.com/Manual/com.unity.netcode.gameobjects.html. Accessed: 2023-05-04.

[20] Unity. Networkanimator. https://docs-multiplayer.unity3d.com/netcode/current/components/networkanimator. Accessed: 2023-05-05.

[21] Unity. Networktransform. https://docs-multiplayer.unity3d.com/netcode/current/components/networktransform. Accessed: 2023-05-05.

[22] Unity. Relay. https://docs-multiplayer.unity3d.com/netcode/current/relay/. Accessed: 2023-02-17.

[23] Unity. System requirements for unity 2021.1. https://docs.unity.cn/2021.1/Documentation/Manual/system-requirements.html. Accessed: 2023-05-03.

[24] Unity. Unity 2021.3.16f. https://unity.com/releases/editor/whats-new/2021.3.16. Accessed: 2023-05-04.

[25] Unity. Unity lobby service. https://docs.unity.com/lobby/en/manual/unity-lobby-service. Accessed: 2023-02-17.

Appendix A

Source code

ClientNetworkTransform

```
using Unity.Netcode.Components;
using UnityEngine;

namespace Unity.Multiplayer.Samples.Utilities.ClientAuthority
{
    /// <summary>
    /// Used for syncing a transform with client side changes.
    /// This includes host. Pure server as owner isn't supported by this.
    /// Please use NetworkTransform for transforms that'll always be owned by the server.
    /// </summary>
    [DisallowMultipleComponent]
    public class ClientNetworkTransform : NetworkTransform
    {
        /// <summary>
        /// Used to determine who can write to this transform. Owner client only.
        /// This imposes state to the server. This is putting trust on your clients.
        /// Make sure no security-sensitive features use this transform.
        /// </summary>
        protected override bool OnIsServerAuthoritative()
        {
            return false;
        }
    }
}
```

OwnerNetworkAnimator

```
using Unity.Netcode.Components;

public class OwnerNetworkAnimator : NetworkAnimator
{
    protected override bool OnIsServerAuthoritative()
    {
        return false;
    }
}
```

RPC Request Structure in Unity Netcode

```
public void Grab(ulong networkObjectId, ulong clientId)
{
    GrabServerRpc(networkObjectId, clientId);
}

[ServerRpc(RequireOwnership = false)]
private void GrabServerRpc(ulong networkObjectId, ulong clientId)
{
    GrabClientRpc(networkObjectId, clientId);
}

[ClientRpc]
private void GrabClientRpc(ulong networkObjectId, ulong clientId)
{
    OnAnyGrabbed?.Invoke(this, new OnAnyGrabbedEventArgs
    {
        grabberNetworkObjectId = EmoteManager.Instance.GetConnectedPlayerById(clientId)
                                 .NetworkObjectId,
        grabbedNetworkObjectId = networkObjectId
    });
}
```

FollowTransform

```
public class FollowTransform : MonoBehaviour
{
     private Transform targetTransform;

    public void SetTargetTransform(Transform targetTransform)
    {
        this.targetTransform = targetTransform;
    }

    public void ResetTransform()
    {
        targetTransform = null;
    }

    private void LateUpdate()
    {
        if (targetTransform == null)
        {
            return;
        }

        transform.position = targetTransform.position;
        transform.rotation = targetTransform.rotation;
    }
}
```

Quick Join (without Relay)

```
public async void QuickJoin()
{
    OnJoinStarted?.Invoke(this, EventArgs.Empty);

    try
    {
        joinedLobby = await LobbyService.Instance.QuickJoinLobbyAsync();

        GameMultiplayer.Instance.StartClient();
    }
    catch (LobbyServiceException e)
    {
        Debug.Log(e);
        OnQuickJoinFailed?.Invoke(this, EventArgs.Empty);
    }
}
```

Quick Join (with Relay)

```
public async void QuickJoin()
{
    OnJoinStarted?.Invoke(this, EventArgs.Empty);

    try
    {
        joinedLobby = await LobbyService.Instance.QuickJoinLobbyAsync();

        // ------- RELAY -------
        string relayJoinCode = joinedLobby.Data[KEY_RELAY_JOIN_CODE].Value;

        JoinAllocation joinAllocation = await JoinRelay(relayJoinCode);

        NetworkManager.Singleton.GetComponent<UnityTransport>()
        .SetRelayServerData(new RelayServerData(joinAllocation, "dtls"));
        // ------- RELAY -------

        GameMultiplayer.Instance.StartClient();
    }
    catch (LobbyServiceException e)
    {
        Debug.Log(e);
        OnQuickJoinFailed?.Invoke(this, EventArgs.Empty);
    }
}
```

Parabolic Trajectory using Mathematical Calculations

```
private void Parabola()
{
    // Progress goes from 0 (startPosition) to 1 (endPosition)
    progress = Mathf.Min(progress + Time.deltaTime * stepScale, 1.0f);

    // Translates Progress value into a parabola
    float parabola = 1.0f - 4.0f * (progress - 0.5f) * (progress - 0.5f);

    // NextPos is obtained by a straight line from the startPosition to the endPosition.
    Vector3 nextPos = Vector3.Lerp(startPosition, endPosition, progress);

    // NextPosition vertical value gets added the arc
    nextPos.y += parabola * arcHeight;

    // Reset the parabola when it has reached the end
    if (progress == 1)
    {
        ResetParabola();
    }
}
```