Overforged : Design and Development of an Indie Simulation Game

Eliseu Camilo Batista

Relatório do projeto em

Design e Desenvolvimento de Jogos Digitais

(2º ciclo de estudos)

Orientador: Prof. Doutor Frutuoso Gomes Mendes da Silva

Covilhã, junho de 2022

Declaração de Integridade

Eu, Eliseu Camilo Batista, que abaixo assino, estudante com o número de inscrição M10664, do mestrado em Design e Desenvolvimento de Jogos Digitais, na Faculdade de Artes e Letras, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o **Código de Integridades da Universidade da Beira Interior**.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assuminado assim na integra as responsabilidades de autoria.

Universidade da Beira Interior, Covilhã, 09/06/2022

(Assinatura)

Acknowledgements

Work developed at ”Instituto de Telecomunicações” under supervision of the Prof. Frutuoso Silva, whom I thank for mentoring and supporting me once more.

Resumo

Overforged é um jogo de simulação situado numa época medieval fictícia. O jogo é inspirado fortemente no título *Overcooked* da desenvolvedora Ghost Town Games, porém neste jogo, ao invés de cozinhar o jogador deve forjar.

O jogador desempenha o papel de dois irmãos ferreiros, de aparência customizável, que são chantageados pelo rei a viajar o mundo.

Neste documento será acompanhado o processo de design dos diversos elementos e mecânicas que compõem o jogo, bem como o desenvolvimento das mesmas. Ao longo deste processo, serão referidos os principais problemas encontrados bem como as estratégias acompanhadas para os ultrapassar.

Durante todo o desenvolvimento foram utilizadas animações oferecidas pela Mixamo, criando manualmente apenas as que a ferramenta não oferece. O processo de desenvolvimento foi realizado no *Unity Engine*, que assenta sobre a linguagem de programação C#.

Desenvolveu-se um jogo composto com mecânicas de cooperação local, vários níveis que podem ser selecionados num mapa mundo e diversos personagens.

Palavras-chave

Unity Engine, *Simulação*, *Jogo Indie*, *Jogo Cooperativo*, C#

Abstract

Overforged is a simulation game set in a fictional medieval era. The game is heavily inspired by the title *Overcooked* by developer Ghost Town Games but in this game, instead of cooking, the player must forge.

The player plays the role of two blacksmith siblings, with customizable appearances, who are blackmailed by the king to travel the world.

This document will monitor the design process of the various elements and mechanics that make up the game, as well as their development. Throughout this process, the main problems encountered will be mentioned, as well as the strategies followed to overcome them.

During all development, animations offered by Mixamo were used, manually creating only those that the tool does not offer. The development process was carried out in the *Unity Engine*, which is based on the C# programming language.

A game made of local cooperation mechanics, several levels that can be selected on a world map, and several characters was developed.

Keywords

Unity Engine, *Simulation*, *Indie Game*, *Coop Game*, C#

Contents

1 Introduction **1**

1.1 Context and Motivation 1

1.2 Objectives and Method 1

1.3 Document Structure 2

2 Related Work **3**

2.1 The History Of Blacksmithing 3

2.2 The Forging Process 3

2.3 Overcooked 2 4

2.4 Forged In Fire 6

3 The Design of Overforged **7**

3.1 Concept 7

3.2 Story 8

3.3 Character 9

3.4 Camera 12

3.5 Controls 13

3.6 HUD & Screens 15

3.6.1 Title Screen 17

3.6.2 Main Menu 18

3.6.3 Pause Screen 20

3.6.4 Loading Screen 21

3.6.5 Other Screens 22

3.7 Level Design 23

3.8 Mechanics 27

3.8.1 Grab And Drop Objects 28

3.8.2 Throw Objects 28

3.8.3 Take Objects From Containers 29

3.8.4 Use Balconies 29

3.8.5 Use The Furnace 30

3.8.6 Use The Anvil and The Saw 30

3.8.7 Use The Quencher 31

3.8.8 Use The Table 32

3.8.9 Deliver Objects 32

3.8.10 Change Character Control 33

3.8.11 Kill and Respawn a Character 33

3.9 Multiplayer 34

3.10 Music and Sounds 35

4 **The Development of Overforged** 37
4.1 Setting Up The Unity Project 37
4.1.1 Setting Up The Input System 37
4.1.2 Setting Up The Devices Logic 40
4.2 Game Bases 42
4.2.1 Player Base 42
4.2.2 Level Management 44
4.2.3 UI Base 45
4.2.4 Game Settings 50
4.2.5 Serialization 54
4.2.6 Loading Screen 55
4.2.7 The Loading Process 56
4.3 Main Menu 58
4.3.1 Settings Screen 60
4.3.2 Customization Screen 62
4.3.3 Controls Screen 62
4.4 World Map 63
4.4.1 World Map Environment 63
4.4.2 The Level Logic 64
4.4.3 World Map Player 66
4.4.4 World Map Interactables 67
4.4.5 World Map UI 70
4.5 Playable Levels 71
4.5.1 Tutorials 72
4.5.2 Time, Score And Oders 73
4.5.3 The Playable Level UI 76
4.5.4 Playable Levels Persistent Data 78
4.5.5 Playable Levels Player 78
4.6 Interactables 82
4.6.1 Movable Interactables 83
4.6.2 Metal 84
4.6.3 Blade 86
4.6.4 Wood 86
4.6.5 Handle 87
4.6.6 Weapon 88
4.6.7 String 88
4.6.8 Static Interactables 89
4.7 Level Specifics 94
4.7.1 Wagon 94
4.7.2 Raft 96

5 Tests **99**
5.1 Preparing The Game For The Tests 99
5.2 Planning The Tests 101
5.3 Tests Results 102
5.4 Tweaking Game 104
5.4.1 Size Of The Game's UI In Level 3 104
5.4.2 Increase Light Emitted By Items 104
5.4.3 Loading the First Level Instead Of The World Map 105
5.4.4 Making The Process of Finding Materials More Explicit 105

6 Conclusions And Future Work **107**
6.1 Conclusions 107
6.2 Future Work 107

Bibliography **109**
References 109

A Game Engagement Questionaire **111**

List of Figures

2.1 Cover of the game Overcooked 2 5
2.2 Forged In Fire Logo 6

3.1 Concept art for the World Map 8
3.2 Concept for the Comic Book Format 9
3.3 Concept of Two Overforged Characters 10
3.4 Concept of The Dash Actions 11
3.5 Concept of The Grab and Throw Actions 11
3.6 Concept of The Character Control Situations 11
3.7 Concept of the Bird View used by the Camera 12
3.8 Behavior of the World Map Camera 13
3.9 What happens when the player moves the left stick to the right with camera related controls 13
3.10 What happens when the player moves the left stick to the right with player related controls 14
3.11 Fingers response and usage 14
3.12 Controls Scheme For Gamepad 15
3.13 Controls Scheme For Keyboard 15
3.14 Concept of the HUD elements 16
3.15 Concept of the Order Panel 17
3.16 Concept of the Title Screen 18
3.17 Concept of the Main Menu 18
3.18 Concept of the Start Game screen 19
3.19 Concept of the Customization screen 19
3.20 Concept of the Settings screen 20
3.21 Concept of The Gamepad and Keyboard controls screens 20
3.22 Concept Of Pause screen 21
3.23 Concept Of Loading screen 22
3.24 Concept Of Level Details Panel 22
3.25 Concept Of The Tutorial Screen 23
3.26 Concept Of Score Panel 23
3.27 Process of forging a blade 24
3.28 Process of making a handle 24
3.29 How to merge the blade and the handle 25
3.30 Results of merging the katana with the guard and the viking axe with the string 25
3.31 Concept Of Level 27
3.32 Concept Of The Player Grabbing An Object From The Ground 28
3.33 The two possible outcomes when the player drops an object. 28

3.34 The two possible outcomes when the player throws an object. 29
3.35 Concept of the player taking an material from the container. 29
3.36 Concept of the player placing and taking an object from the balcony 30
3.37 Concept of the player heating a metal on the furnace. 30
3.38 Concept of the player shaping metal on the anvil. 31
3.39 Concept of the player quenching a blade. 31
3.40 Concept of the player merging two objects on the table. 32
3.41 Concept of the player quenching a blade. 32
3.42 What happens when the player tries to change the controlled character when playing solo. 33
3.43 What happens when the player tries to change the controlled character when playing in co-op. 33
3.44 What happens when the player is run over. 34

4.1 Project creation screen and the chosen template 37
4.2 The New Input System Package 38
4.3 Representation of the InputActions asset 38
4.4 Gameplay Input Map in the InputActions Asset 39
4.5 Devices Logic Used In The Game 41
4.6 POLYGON MINI - Fantasy Characters Pack from Synty Studios 42
4.7 Fantasy Characters Prefab 43
4.8 Animator Locomotion State 44
4.9 Slider UI Component 46
4.10 Slider UI Component 47
4.11 EventTrigger component of the button 48
4.12 Button Events Logic 49
4.13 High Quality Graphics Stettings 52
4.14 Overforged Audio Mixer 53
4.15 Example of an Loading Tip 56
4.16 Loading Screen Behavior 57
4.17 Loading Screen 58
4.18 Main Menu Structure 58
4.19 TitleScreen menu from the Main Menu 59
4.20 Neighbours of the Customization button 60
4.21 Main Menu Settings Screen 61
4.22 Customization Menu 62
4.23 Controls Menu 63
4.24 POLYGON MINI - Fantasy Pack from Synty Studios 64
4.25 Overforged world map 64
4.26 Marking an Object as Navigation Static 65
4.27 World Map Walkable Unit 66
4.28 World Map Path Made Of Walkable Units 66
4.29 World Map Player 67

4.30 Ladder Logic 68
4.31 Player animations played when climbing the ladder 69
4.32 Boat Mode of the World Map Player 70
4.33 UI Panel With Level Information 70
4.34 World Map Pause Menu 71
4.35 World Map Pause Settings 71
4.36 Tutorial Screen 73
4.37 Level Timer Coroutine Logic 74
4.38 Orders Logic In The Playable Levels 76
4.39 Playable Levels HUD 77
4.40 Playable Levels Pause Menu 77
4.41 Playable Levels Score Panel 78
4.42 Calculation Of The Movement Vector Based On The Camera 79
4.43 Ground Check Using Raycast 80
4.44 POLYGON - Particle FX Pack from Synty Studios 81
4.45 Dash Action 81
4.46 Interactable Lit and Unlit 83
4.47 Movable Interactable Being Carried By The Player 84
4.48 The Three States Of A Metal 85
4.49 Metal Before And After The Hammering Process 85
4.50 Blade Before And After The Quenching Process 86
4.51 Wood Before And After The Shaping Process 87
4.52 Example Of An Handle 87
4.53 Example Of An Weapon 88
4.54 Example Of An String 88
4.55 Interactable Placed In A Balcony 90
4.56 Anvil, a ActiveBalcony 91
4.57 Furnace, a PassiveBalcony 91
4.58 Example of an MaterialSource 92
4.59 Quenching Quick Action 93
4.60 Table, an Quick Action Balcony 93
4.61 Delivery Point 94
4.62 Wagon Behavior 95
4.63 Wagon, The Level 2 Delivery Point 96
4.64 Rafts in Level 3 97

5.1 Values related to player locomotion represented in the editor 100
5.2 Knife Weapon Hierarchy 100
5.3 Likert Scale applied to a question of Nicola Whitton questionaire. 102
5.4 Example of the graph generated by the forms for one of the questions. . . . 103
5.5 New camera view in level 3, fixing the User Interface (UI) overlapping issue. 104
5.6 Example of one of the added panels. 105

Code Snippet List

4.1 Creating an instance of GameInputs and listening to the Dash input action 39
4.2 Binding a Vector2D action to a function 40
4.3 Listenning to the onDeviceChange callback 40
4.4 Class ”Button Neighbor” used to configure the button neighborhood system. 49
4.5 Creating the GameSettings instance 50
4.6 Changing The Music Volume 54
4.7 Surrogate for the GameSettings class 54
4.8 Loading Tip Class containing the different attributes of an loading tip. . . . 56
4.9 Subscribing to button events 59
4.10 Subscribing to ”Music Volume” button events 61
4.11 Class ”Tutorial” containing the tutorial information. 72
4.12 Class ”Order” containing the different attributes of an order. 74
4.13 Changing the Rigidbody Velocity 80

Acronyms List

UI User Interface

HUD Head-Up Display

SFX Sound Effect

URP Universal Render Pipeline

HDRP High Definition Render Pipeline

Chapter 1

Introduction

Video games play a huge role in the entertainment industry and their development is not done exclusively by companies but also by small groups of people or even a single person.

While a company usually has tools that help the development process and also people whose sole focus is to improve these tools, an independent developer usually resorts to free tools. Of these tools, the most popular are engines such as Unity and Unreal Engine. Despite the limitations of these engines, they allow the developers to build complete and polished games.

One of the biggest challenges in game programming is the programming of multiplayer, especially network multiplayer. When having more than one player, the developer will have to take into account many more variables and situations, therefore more care and attention are needed.

1.1 Context and Motivation

This project is part of a 2nd-year curricular unit called "Thesis, Project or Internship" of the Design and Development of Digital Games Master's Degree. The project consists of the design and development of a forge simulation video game like Overcooked 2.

These days, almost every game has some aspect of network multiplayer, letting offline mode and local multiplayer mode out of use. An online-only multiplayer game requires that two players who want to play together must each own a copy of the game, which can sometimes be a reason why a game does not have a larger number of active users. To get around this problem, the developed game will be a local multiplayer game, and in the future, we also aim to add the network multiplayer feature.

The developed game is inspired by Overcooked 2, a game that can be played by both casual players and competitive players, with no age restriction.

Knowing how to develop and create in Unity Engine using the C# language is a very frequent requirement for anyone looking to work in the video game industry as a developer. This project aims to strengthen the knowledge of Unity Engine and C# and to design and develop a game that serves as proof to the industry of the developer's capabilities with the Unity tool.

1.2 Objectives and Method

The main goal of this project is to **develop a simulation game like Overcooked 2** with local multiplayer, but in this game, the player must forge in several levels instead

of cooking, and for that, it will be necessary to research about the area.

Just like Overcooked does with the cooking process, the forging process will not be very precise and will only have a few essential steps. The game is not intended to represent a **realistic method of forging**, but **just a fun method**.

In order to avoid major changes to the game, it will be designed to work in multiplayer from the start.

The development process will start by creating the input system and player bases and levels, and these will be developed so that they are reusable. Then we create the game's main menu, where the game settings and character customization screens will be created. Next, the world map will be developed along with the data persistence system, so that players can save and upload their progress.

When the map is functional, the playable levels will be implemented, where players will be able to move objects, combine them and complete their tasks.

Lastly, bugs will be fixed and multi-player testing will be performed.

1.3 Document Structure

This document is made of 3 more chapters in addition to the current one:

Chapter 2 – **Related Work** – presents a summary of the history and process of smithing, as well as the video games that were a great inspiration to this project;

Chapter 3 – **The Design Of Overforged** – analyzes the stages of the design of the game;

Chapter 4 – **The Development Of Overforged** – follows the development phases of the game;

Chapter 5 – **Tests** – presents the performed tests and their results;

Chapter 6 – **Conclusions and Future Work** – presents the main conclusions obtained by developing the project, as well as the plans for Overforged.

Chapter 2

Related Work

In this chapter, we discuss the history and process of smithing. Finally, we analyze the game Overcooked 2, which is the main inspiration for the game reported in this document.

2.1 The History Of Blacksmithing

Forging is the process of **shaping metal using techniques** that generally consist of submitting metal to pressure and forces.
According to the article ”History of Blacksmithing” on the Industrial Heating website (IndustrialHeating, 2011), we can say that the first smiths were those who heated iron in bonfires, our ancestors discovered that metal was moldable when heated to high temperatures.

It was only later, with the industrial revolution, that the smiths began to specialize in different categories, such as the whitesmith, who worked with lead, and the Blacksmith, who worked with iron.

Later, by the 16th century, smithing became an art, and the smiths were more concerned with the artistic component rather than the usefulness of the object. Nowadays, the demand for stationery items is due to the uniqueness of the objects and not to their usefulness. Today's blacksmiths use more sophisticated equipment, but still, many prefer to forge the old-fashioned way, using mostly tools such as a hammer, anvil, and tongs.

2.2 The Forging Process

There are several methods and techniques for forging, which vary with the metal and the temperature at which they are made.

According to the article ”Understanding Metal Forging Processes, Methods, and Applications” on the TFGUSA website (TFGUSA, 2020), before starting the forging process, it is important to choose the technique to use, as each technique brings a set of advantages and disadvantages that must be taken into account. Techniques mostly differ in the way the metal is moved and deformed as well as the used tools.

In the most general case, to deform the metal, it is first heated to high temperatures, between 500°C and 1300°C, and then deformed. According to the article in question, there are four standard tools used in the process of deforming metal:

- **Hammers:** This tool is used to repeatedly deform metal by impact;
- **Presses:** Slowly deform the metal through constant vertical pressure applied to it;

- **Upsetters**: Similar to Presses, but these apply force horizontally;

- **Ring Rollers:** Used to easily produce rings.

When molding the metal we also compress it, removing impurities and gaps between the material, resulting in a stronger and more resistant metal. Finally, the cooling process is the last essential step of the forging process. According to the ”Types Of Quenching Process for Blacksmithing” guide posted on the ”Working The Flame” website (Flame, 2020), cooling can be done naturally by letting the metal cool in the open air, or using a liquid such as water and oil. There are 3 main methods for cooling metal and a 4th uncommon method:

- **Outdoor cooling** is affordable, does not have a major impact on the metal structure, and is therefore generally not the first choice.

- **Water cooling** is the most common, it is affordable, quickly cools the metal, and greatly increases the hardness of the material. This process may, however, deform the metal or make it too brittle if done incorrectly.

- **Oil-cooling** is a more expensive process, it quickly cools the metal and increases its hardness. The increase in hardness is less than cooling with water, but the likelihood of deforming the metal or brittleness is small.

- There is also a fourth method, which consists of **mixing oil, water, and salt**, and it is the most effective method to successfully increase the hardness of a material. This method is, however, quite polluting and expensive.

2.3 Overcooked 2

Overcooked 2 (Team17, 2018) is a simulation kitchen game developed by Ghost Town Games and published by Team 17, where the player plays the role of one or more chefs and must prepare meals in the most diverse environments, within a limited time. The game was released in 2018 for Xbox One, Playstation 4, Nintendo Switch, and computer. The figure 2.1 represents the cover of the game Overcooked 2.

Figure 2.1: cover of the game Overcooked 2. Image from
https://www.epicgames.com/store/pt-BR/p/overcooked-2

The Game can be played by up to 4 players locally or online, cooperatively or competitively. The game has a story mode, and an arcade mode. In story mode players must cooperate to complete levels, while in arcade mode players can cooperate or compete against each other.

In the game, players play the role of customizable chefs and must **prepare the required dishes** by collecting ingredients, preparing them, and combining them until they get the desired result. **The kitchens** used by the player **are usually a chaotic environment**, with movable platforms and cars that can run over players.

Players can throw some objects to specific places or other players to save time.

As can be read on the Metacritic website (Metacritic, 2018), the game was positively received, with a current Metascore of 81. According to reviews, especially William Thompson's review writing for HookedGamers (Thompson, 2020), Overcooked 2 is a game that can be enjoyed by all types of players, from casuals who just want to have fun, to competitive ones looking to get the best scores, "a great game for the whole family".

Overcooked 2 is a game that encourages cooperation and is challenging for those looking for a challenge, and relaxing for those looking to relax. The aim is to develop a game like this that offers a challenging experience for those who want to get the highest scores and a relaxing experience for those who just want to have fun. As in Overcooked 2, simple mechanics and a simple control scheme will be implemented, so that even the most inexperienced person can play. In Overcooked 2, cooking in chaotic settings, where in reality you could never cook, is what makes the game unique and different, and that same element will be included in Overforged.

2.4 Forged In Fire

Forged In Fire, according to Wikipedia, is an American television series, produced by *Outpost Entertainment*. In each episode of the series, 4 smiths compete against each other in 3 different rounds, and in each round one of them is eliminated, leaving only the winner who earns the title of ”Forged In Fire Champion” (Wikipedia, 2022a). The figure 2.2 illustrates the logo of the television series *Forged In Fire*.

Figure 2.2: Forged In Fire Logo. Image from https://en.wikipedia.org/wiki/Forged_in_Fire

In the first round of the competition, the smiths must forge a blade, according to the requirements imposed by the juries, from a set of materials and tools available on the spot. The round usually lasts 3 hours, during which time the smiths must collect the necessary metal, heat and shape that metal and finally sharpen that blade. At the end of the first round, the judges evaluate the blades of the 4 participants and choose one to be eliminated.

In the second round, smiths must produce a handle for their blades, and may also correct any flaws and aspects of the blade forged in the first round. In this round, smiths look for the ideal material for their handle and using a technique of their choice, attach the handle to their blade. This round usually lasts 1 hour, and at the end, the blades will be tested for strength, sharpness, and resistance. The tests are varied and depending on the results, another smith is eliminated.

The remaining two smiths will, in the third round, forge a more complex weapon in their forges. Participants will have a week to forge the weapon, which will be tested in strength, sharpness, and endurance. According to the test results and parameters of the weapons presented, one of the participants will be chosen as the winner and will earn the title of ”Forged In Fire Champion”.

Forged In Fire was the main inspiration for the game’s theme *Overforged*.

Chapter 3

The Design of Overforged

Before starting the production of a game, it is important to plan the aspects that make up the game, to develop something fun and concise. This planning process is called **Game Design**.

The design of the game *Overforged* will follow the work *Level Up! The Guide To Great Videogame Design* by the author Scott Rogers (Rogers, 2010). This work was written by the author to share his practical knowledge with professionals working with video games, aspiring game designers, game design students, and anyone who loves video games. The work is divided into several levels, starting at level 1 to level 17, offering some bonus levels. Each level addresses an aspect that the author considers essential in the game design process.

The Design of Overforged will be done following the book level by level, only skipping those that do not apply to this game.

3.1 Concept

The *Level 1* of the work is a brief presentation of the video game industry, history and development process and the reader is told that the level is intended for newbies only, and is therefore not important for the game design *Overforged*.

Level 2 talks about ideas, how to get ideas and inspiration, and give the reader tips on the *Brainstorming* process. While this chapter guides the reader to find a ”unique” idea of their own, the idea of this project is to replicate an existing game, *Overcooked*, changing only the theme of the game, and therefore, the follow-up to the book will begin only at *Level 3*.

The key idea of the game is:

Two blacksmiths travel the world, forging weapons in the most diverse environments. The two blacksmiths must work together to complete the forging process within the time limit.

The game will have a world map, where the player can choose the levels and each level will have a set of 3 stars, which will require different scores. Within the level, the player will have a few minutes to deliver the Orders presented to him, and for that, he must work as a team with the two characters present in the level. If the player is playing alone, he can switch the character he controls, otherwise, each player controls one character. The game will be limited to two players. Figure 3.1 illustrates the concept art for the world map.

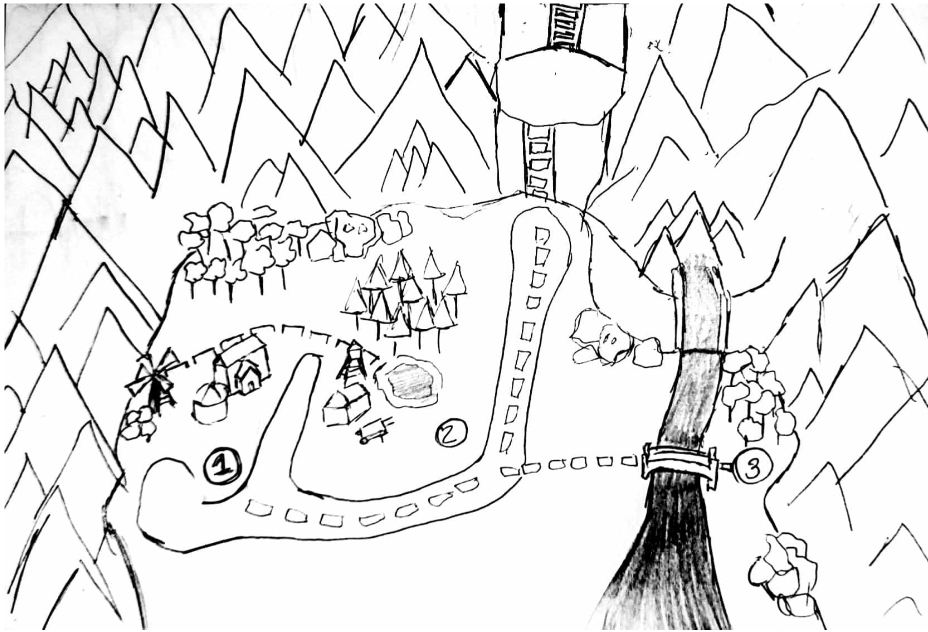

Figure 3.1: Concept art for the World Map

3.2 Story

Following the levels of the book, the next level is level 3, with the development of the game's story. According to the author, when developing a story for a video game, there are 3 types of players to consider (Rogers, 2010, p. 46):

- Players that care about the story as it happens;
- Players that care deeply about the story;
- Players that don't care about the story.

Satisfying the 3 types can be a challenge and therefore Scott Rogers advises that **a story must be made to complement the gameplay**, and not the other way around (Rogers, 2010, p. 46). In *Overforged*, the player moves from level to level on a world map, and the levels consist of forging items within the time limit, so we want a story that justifies the following points:

- Why do the characters travel the world to forge?
- Why are the characters forced to forge within a time limit?
- And why do the two of them always go together?

A story idea that might answer these points would be:

In a village lived two brothers, sons of one of the best blacksmiths in the kingdom. This kingdom has always lived peacefully with the neighbor kingdoms, but after the death of the former king, his son, who succeeded the throne, guided by greed, sought to

conquer all neighboring kingdoms. During the war, the brothers' father worked until he was old and tired, and the new king, seeing that he had lost one of his best blacksmiths, decided to use him as a hostage, to threaten his sons and make them work for him. The sons then travel to the various battlefields, at the behest of the king, where they must exhaustively forge, in the hope of eventually recovering their father.

This idea leaves, of course, many questions unanswered and therefore, to decide how much the story needs to be developed, it is first necessary to choose the type of audience for this game. According to the author, to **satisfy players** who **care deeply about the story**, the best method is to **provide optional details**, such as collectibles, that **complement the story**, so we don't **bore the other two types of players** (Rogers, 2010, p. 46). In *Overforged*, there is no good fit for a collectible or item system, and therefore, the game **focuses on two types of players: those who have no interest in the story, and those who enjoy the story as it happens**.

For these two types of players, the story depicted above is a sufficient starting point, offering a little more with each played level. Making cutscenes can be a more time-consuming and demanding process for the developer and therefore the comic book format was chosen to present the story. Figure 3.2 represents a concept for the comic book format.

Figure 3.2: Concept for the Comic Book Format

3.3 Character

The next level is level 4, whose topic is Game Design Documents, where the author teaches the different types and how to write one (Rogers, 2010, p. 57). Then there is level 5, where the author informs that there are 3 elements that, according to Scott Rogers, are the fundamental pillars of a game, and that should be established as soon as possible, as

any change to them implies the review and refactoring of many mechanics (Rogers, 2010, p. 83). These 3 elements are:

- **Character**;
- **Camera**;
- **Control**.

In *Overforged* the player can customize the appearance of his characters by choosing a character from a predefined collection.

According to Scott Rogers, one of the first steps to be able to create the main character is thinking about their personalities and traits (Rogers, 2010, p. 84). In the case of the *Overforged* game, we want the different pickable characters to have different traits, some of them appear to be brave, and others cowards. But despite their personality differences, they all have something in common, larger head size and smaller body size.

As it is intended to offer male and female characters, who appear to have different personalities and origins, it makes no sense to name the characters so the two will be known as siblings. Figure 3.3 shows a concept of two characters for Overforged, one male and one female.

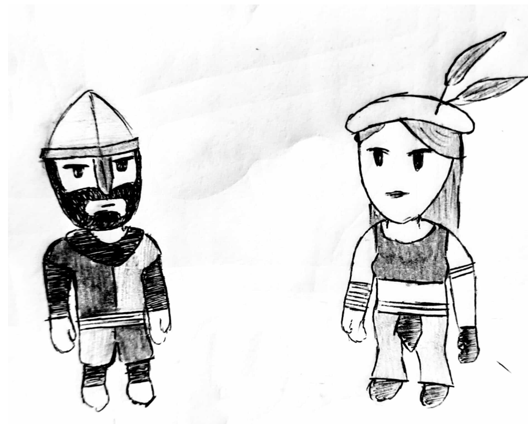

Figure 3.3: Concept of Two Overforged Characters

With the visual component of the characters done, it is necessary to focus on their gameplay metrics and mechanics. As in *Overcooked*, the **player cannot jump or sprint**, there are **only two movement mechanics, the normal movement and the dash**. The movement must be at an accelerated pace so that the character can move quickly between the different elements that make up the level. For the dash, if we use the character's body width as a measure, the dash must move the character 5 times the dimension of its body. Figure 3.4 illustrates the concept of the dash action.

Figure 3.4: Concept of The Dash Actions

The characters can also **grab, drop and throw objects**, which is extremely useful for moving essential items to complete the level. Figure 3.5 represents the concept for the grab action, represented in the figure by the numbers 1 and 2, and the throw action, represented by the number 3.

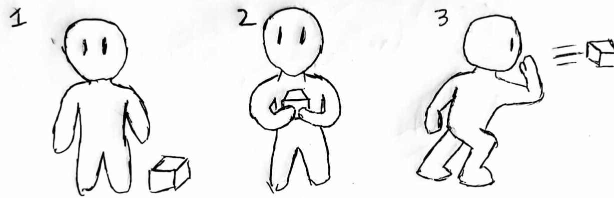

Figure 3.5: Concept of The Grab and Throw Actions

Since *Overforged* is a game that works both as a single-player and multiplayer, there are some disadvantages for the player who plays alone, as he can only control one character. So that the **player who plays alone can have a fairer experience**, the possibility for the player to **switch control between the two characters** will be implemented. Control will be represented by a sphere above the head, which will be red for the character controlled by player 1, blue for the character controlled by player 2, and non-existent if the character is not being controlled. Figure 3.6 shows three characters and the three different control situations. In situation 1, the character is being controlled by player one, in situation 2, it is being controlled by player 2, and in situation 3, it is not being controlled by any player.

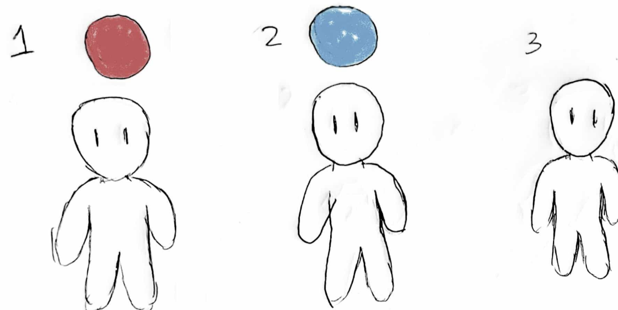

Figure 3.6: Concept of The Character Control Situations

3.4 Camera

The next level is level 6 where the author talks about another of the 3 fundamental pillars, the camera (Rogers, 2010, p. 121). There are several types of cameras used in video games, but almost all of them can fall into two groups, static cameras and dynamic cameras.

Static cameras do not move and always fix the same point. This type of camera, although simple, has advantages that the dynamic camera does not offer. With a static camera, we know exactly what the player will observe, and therefore, we can work on the elements that make up this scenario.

The movable cameras are the type that can be moved by the player or by the game itself, and in this case, it is necessary to pay attention to some things. A dynamic camera can be controlled by the player or the game, and if controlled by the player, there are no guarantees that the player will see what we want him to see, and no guarantees that the camera will not be positioned at unexpected angles.

There are, of course, types of cameras that are a mixture of these two groups. Before moving on to the design of other aspects of the game, the author lists, at Level 6, three types of cameras that we can choose (Rogers, 2010, p. 133), namely:

- Let the player control the camera;
- Do not let the player control the camera;
- Let the player control the camera sometimes.

In *Overforged*, being a game inspired by *Overcooked*, the gameplay mechanics will require the player to **observe all of the environment** during the course of the level, and to prevent the player from getting confused, the camera must **maintain the same position and rotation** throughout the entire level.

We chose **not to let the player control the camera**. Similar to the game *Overcooked*, the camera will observe the player from an angle that the author calls *Birdseye view*, in which the camera's view resembles the view that a bird would have if it were looking down from the sky (Rogers, 2010, p. 145). Figure 3.7 illustrates the bird view used by the camera.

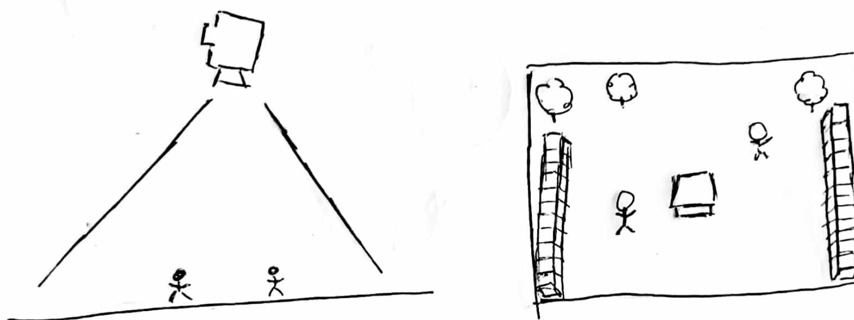

Figure 3.7: Concept of the Bird View used by the Camera

However, **on the world map, things are different**, the camera is still not controlled by the player, but it is not static. **It will follow the player as he moves from level to level**, assuming a predetermined angle that shows the intended scenery elements. Figure 3.8 represents the behavior of the world map camera, where it is visible that when the player moves between levels, the camera follows the player in a predetermined direction.

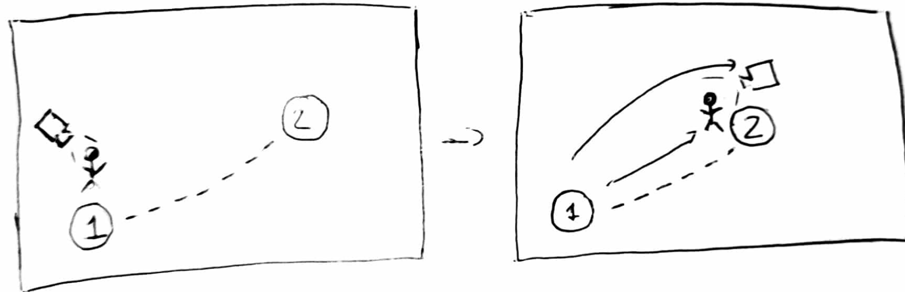

Figure 3.8: Behavior of the World Map Camera

3.5 Controls

Following the levels of the book, the next level is level 7 in which the author talks about the last of the 3 fundamental pillars, the controls (Rogers, 2010, p. 153). According to Scott Rogers, controls can be character-related, or camera-related (Rogers, 2010, p. 164).

When relative to the camera, the controls change and adjust according to the direction the camera is looking, that is, if the player moves the left stick to the right, the character will move to the right of the camera, as illustrated in figure 3.9.

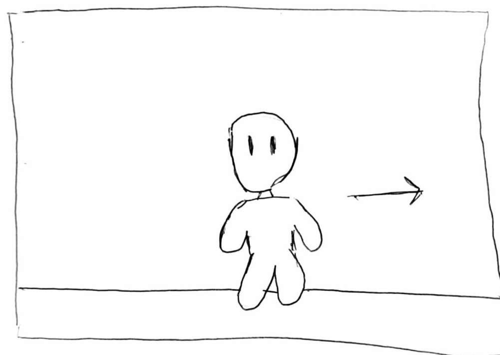

Figure 3.9: What happens when the player moves the left stick to the right with camera related controls

When relative to the character, if the player moves the left stick to the right, the character will move to its right, regardless of the camera view. If the camera is inverted, in this controls mode, when trying to move to the right, the player will go to the left, as illustrated in figure 3.10.

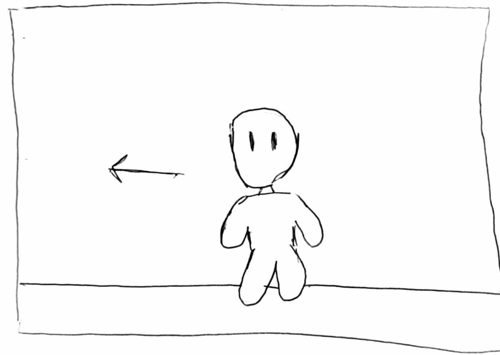

Figure 3.10: What happens when the player moves the left stick to the right with player related controls

As in *Overcooked*, in *Overforged* **the controls are relative to the camera**, so when the player moves the left stick to the right, the character will walk to the right side of the map, without causing any confusion.

Another concern to have with the controls is the ergonomics and the player's response time to the game's control scheme. To define the control scheme, it is necessary to take into account the response and use of the fingers, illustrated in figure 3.11, from the book *Level Up! The Guide To Great Videogame Design* by the author Scott Rogers.

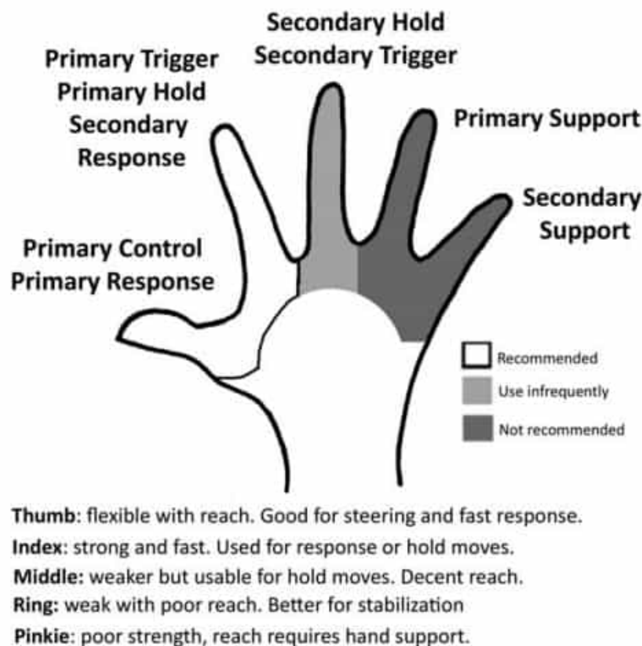

Figure 3.11: Fingers response and usage. Image from the book *Level Up! The Guide To Great Videogame Design* by the author Scott Rogers (Rogers, 2010, p. 157)

In figure 3.11, we observe that, according to the author, the fingers with the fastest response and most frequent use are the thumb and index, followed by the middle. The other two fingers (the ring and the pinkie) should only be used for support and stabilization. Therefore, for *Overforged*, **game actions that require a quick response**, such as *grab*, *drop*, and *shoot* will be assigned to the right hand thumb. The *dash* action will

be assigned to both the thumb and the index of the right-hand, with the player being able to use whichever he is most comfortable with. The *movement* will be performed by the left-hand thumb. Figure 3.12 represents the control scheme for the gamepad.

Figure 3.12: Controls Scheme For Gamepad

Although players are **expected to play with a Gamepad**, **player 1 can play with the keyboard**, player 2, however, will always have to play with a gamepad. When choosing the keyboard's control scheme, we tried to make the keys intuitive and common. In many games, *movement* is almost always assigned to the W, A, S, and D keys, and *interaction* is almost always assigned to the E key, F key, or the Spacebar. Finally, actions like *dash* are often assigned to the left shift or Spacebar. Bearing this in mind, the keyboard controls scheme was defined as shown in figure 3.13.

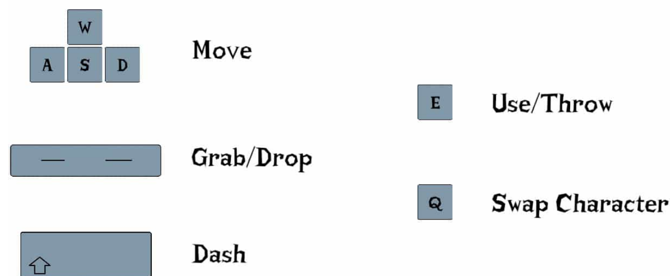

Figure 3.13: Controls Scheme For Keyboard

3.6 HUD & Screens

Next, in Scott Rogers' work, we have level 8, which deals with visual languages, such as the HUD, the different screens, and icons (Rogers, 2010, p. 171). In video games, from the oldest to the most recent, a Head-Up Display (HUD) has always been present. The HUD is, according to Scott Rogers, the most effective means of communicating with the player and is used to communicate information and provide directions to the player. The HUD is composed of visual elements such as text and images (Rogers, 2010, p. 171).

When designing the HUD, the placement of the HUD elements is important, and they should avoid being placed in areas that obstruct the game view. For instance, placing a HUD element in the center of the screen makes no sense in most situations because it will obstruct the area where the action takes place. The corners of the screen are often the most common place to place HUD elements, and for good reason, they are the points furthest from the center of the screen.

In *Overforged*, the player must forge within a time limit and try to get a high score, so the player **must know how much time** he has left and **how many points** he has. It is equally important that the player knows **which objects** he has to forge (orders), their ingredients, and the time left for each one. To inform the player of this information, the following elements will be placed on the HUD:

- Score Panel - lower left corner;
- Time Panel - lower right corner;
- Orders Panel - upper left corner.

Figure 3.14 shows the HUD concept, with the score panel in the lower-left corner, the time panel in the upper right corner, and the orders in the upper left corner.

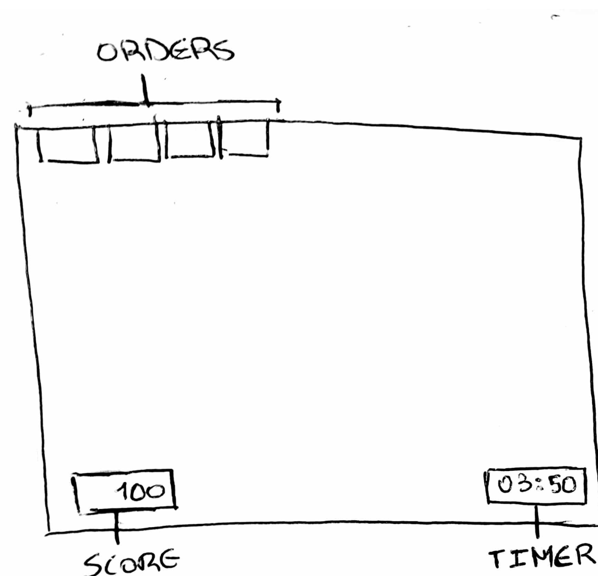

Figure 3.14: Concept of the HUD elements

To **keep the pace of the game faster**, the player must be able to **focus on more than one Order at the same time**, so the Orders panel will display up to 4 Orders at most. More than 4 orders can be too much information and confuse the player.

In the Orders panel, each order will have an image of the object to forge, with a progress bar behind it, which represents the time remaining and the ingredients needed below as illustrated in figure 3.15.

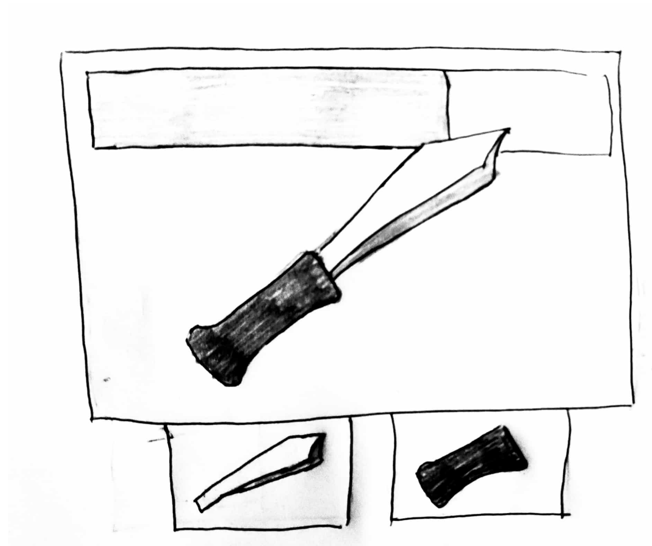

Figure 3.15: Concept of the Order Panel

In addition to HUD, other screens are equally important, for example:

- **Title Screen**;
- **Main Menu**;
- **Pause Menu**;
- **Loading Screen**;

In addition to the ones mentioned above, there are others, but these are the ones that will be present in *Overforged*.

3.6.1 Title Screen

The Title Screen is, after the copyright screens and logos, the first screen to be seen by the player and is, according to Scott Rogers, important to establish a mood (Rogers, 2010, p. 187).

In the case of Overforged, the Title Screen will consist of the name of the game, stamped on 2 metal gates. These are gates to a forge that are closed. When the player presses either the Start button or the Enter key, the gates open, and the player is taken to the main menu. Figure 3.16 illustrates the Title Screen concept.

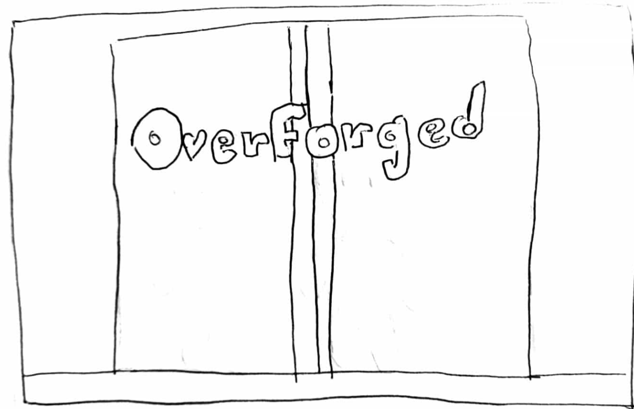

Figure 3.16: Concept of the Title Screen

3.6.2 Main Menu

Instead of setting the mood in the Title Screen, in *Overforged*, this is **done in the Main Menu**. After the doors open, the forge will be visible in the menu, with some materials and tables. Between the doors at the entrance to the forge will be the two playable characters.

In the Main Menu, there are four buttons, one to start the game, one to customize the two characters, one to change the game settings, and the last one to view the controls. Figure 3.17 represents the Main Menu concept.

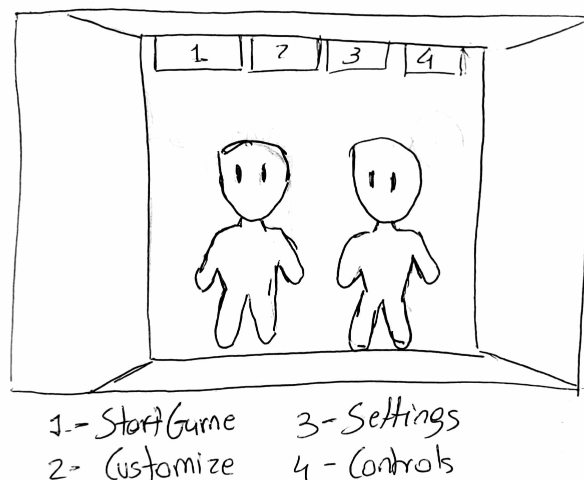

Figure 3.17: Concept of the Main Menu

When choosing the Start game option, if the player has never played, then the game will start immediately. If the player has played before, there will be two buttons, one to start a new game and another to load the saved data. Figure 3.18 shows the Start Game screen concept.

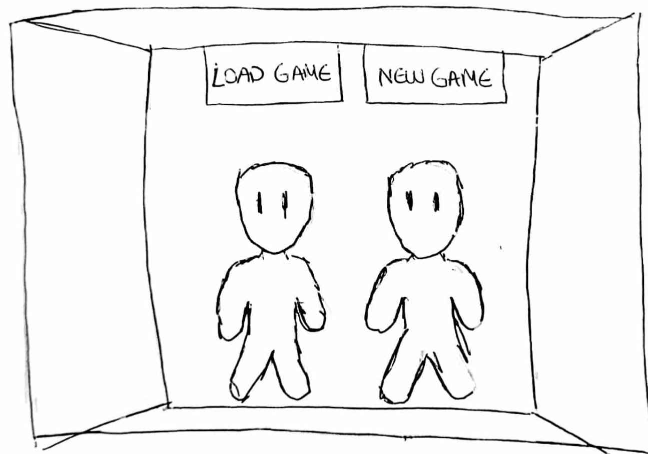

Figure 3.18: Concept of the Start Game screen

When choosing the customize character option, the player will have two buttons at the top, one to save the changes made and another to discard them. Each of the characters will have an arrow button on their left and right, and each pair of buttons will be responsible for changing the character models as shown in figure 3.19.

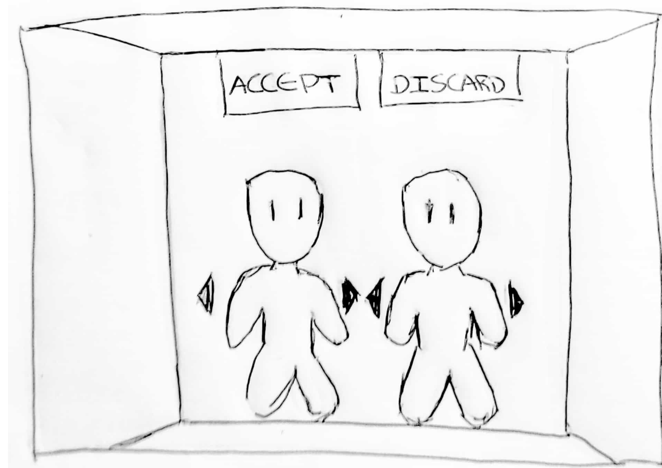

Figure 3.19: Concept of the Customization screen

When choosing the Settings option, the player can change the some of the graphic and audio settings, like:

- Music Volume;
- Sound Effects Volume;
- Screen Resolution
- Fullscreen Mode;
- Graphics Quality;
- Player 1 Device.

The music and Sound Effect (SFX) volumes will be represented by ten bars, which are activated and deactivated depending on the volume. The fullscreen will be represented by a toggle and the rest will be represented by text. Finally, the player will have 2 buttons to discard or save the changes made. Figure 3.20 illustrates the Settings screen concept.

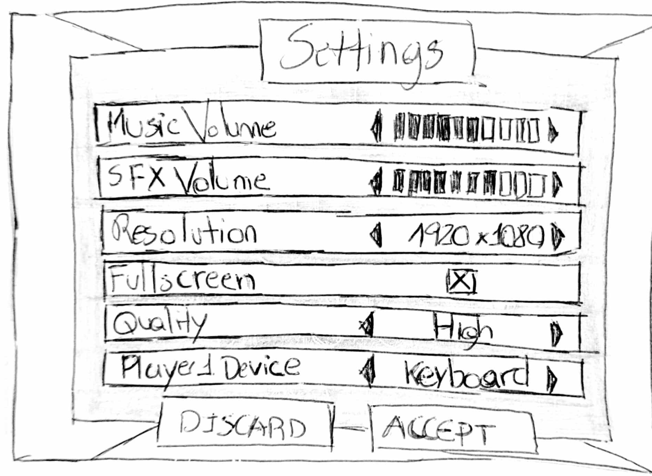

Figure 3.20: Concept of the Settings screen

When choosing the Controls option, the controls for Gamepad will be displayed in a panel. On the panel will be two buttons, one to return to the Main Menu and another to change the controls shown between Keyboard and Gamepad. Figure 3.21 represents the concept of the gamepad controls screen (left) and the keyboard controls screen (right).

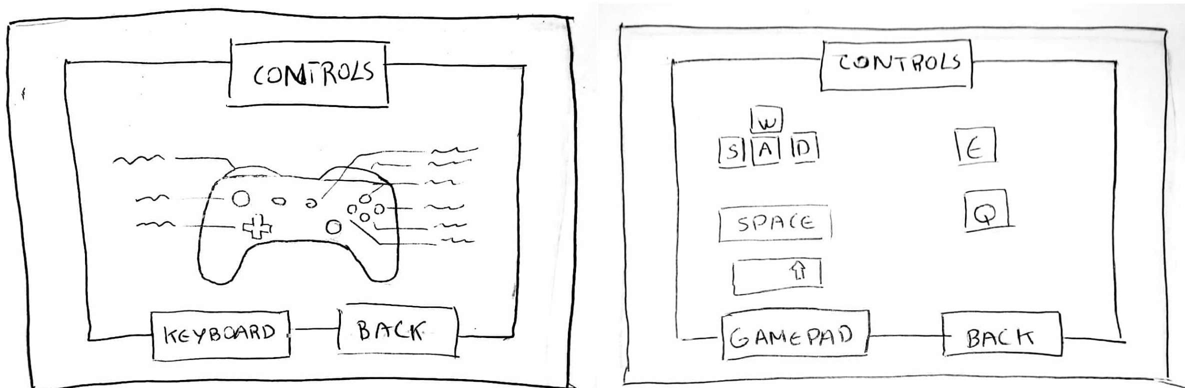

Figure 3.21: Concept of The Gamepad and Keyboard controls screens

3.6.3 Pause Screen

The Pause Screen exists to do what its name implies, offering a pause to the player. However, as Scott Rogers states, this screen can be used to do more than just a pause. For instance, it can be used to save the game, use cheat codes, change the game settings, and exit the game (Rogers, 2010, p. 190).

In *Overforged*, the **Pause Screen** will be used so that the **player can take a break**, change the game settings and exit the game. In the game levels, the exit option should take the player to the world map, and when on the world map, the option should take the player

to the Main Menu. These three options will be displayed as buttons arranged vertically in a panel. Figure 3.22 shows the concept of the pause screen.

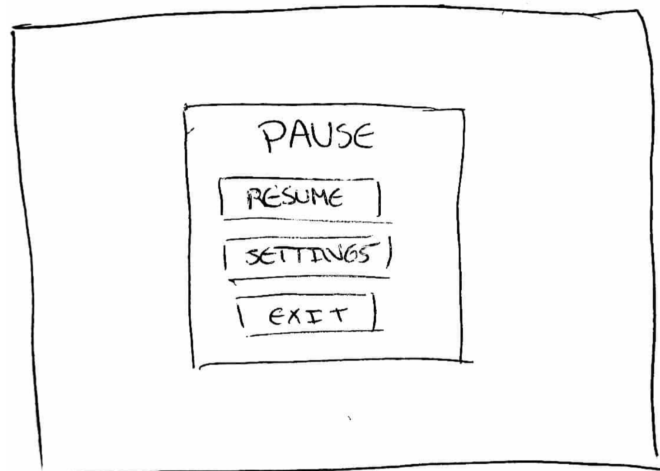

Figure 3.22: Concept of the Pause Screen

3.6.4 Loading Screen

The Loading process is essential in video games to load game models and components. This process can sometimes be too long, and when that happens, the **loading screen works like a theater curtain**, while the staff prepares the scene, the curtains close, and when the scene is ready, the curtains open.

For the players, a loading screen is usually something unpleasant and annoying, but this doesn't necessarily have to be the case. In his book, Scott Rogers lists several ways to make a Loading Screen more immersive and interesting (Rogers, 2010, p. 192), including:

- Show Concept Art;
- Provide tips on gameplay and control;
- Display the game map.

In *Overforged*, the Loading Screen will be used essentially to **provide tips on gameplay while the player waits for the scene to load**. In the loading screen, there will be an image that occupies a large part of the screen, with a representative content of the tip in question, and below that image will be the descriptive text of the tip. Above the image will be the text loading, so that the player can immediately identify that it is a loading screen. Figure 3.23 illustrates the concept of the loading screen.

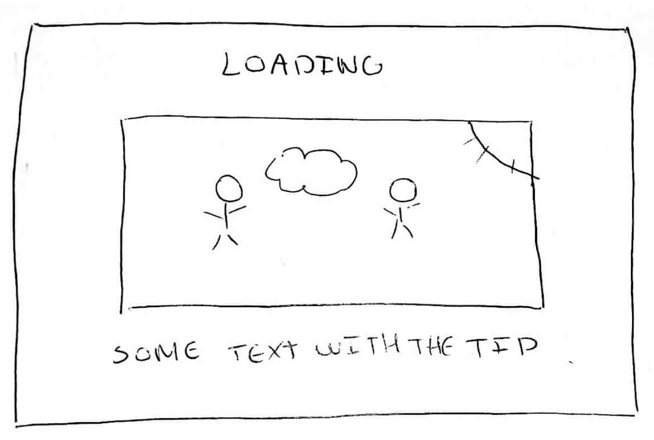

Figure 3.23: Concept of the Loading Screen

3.6.5 Other Screens

There are still some UI elements that are present in the game. On the world map, there is a panel with details about the level in which the player is, in this panel, there is an illustrative figure of the level, the scores needed to obtain the respective stars, and the highest score obtained by the player as illustrated in figure 3.24.

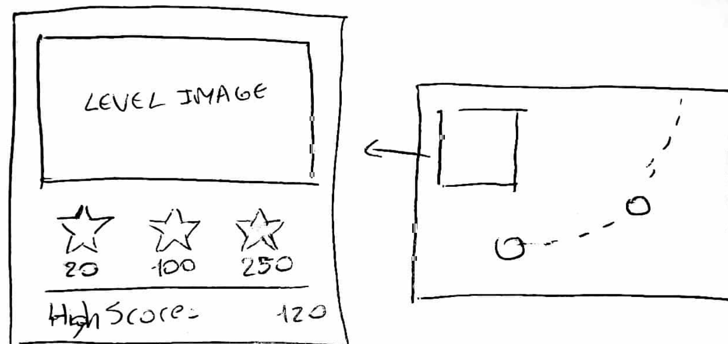

Figure 3.24: Concept Of Level Details Panel

When **starting a level**, a screen will be displayed to **teach the player how to forge** the weapon corresponding to the level. The screen consists of a panel with the title, illustration, and description of the tutorial. Below will be two buttons, one to go back and one to advance between tutorials. In the last tutorial, the forward button is replaced by the close button. Figure 3.25 illustrates the concept of the tutorial screen.

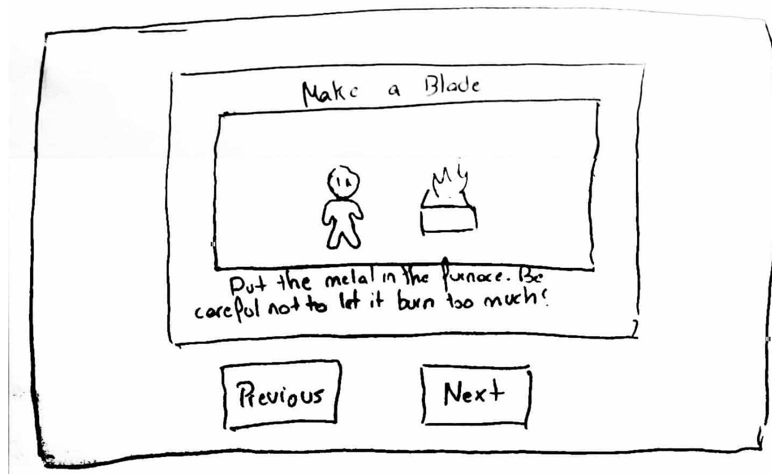

Figure 3.25: Concept Of The Tutorial Screen

When finishing a level, the player is directed to a screen where there is a panel, with the number of stars obtained, the number of orders delivered and failed, and two buttons, to return to the world map and restart the level. Figure 3.26 shows the concept of the score panel.

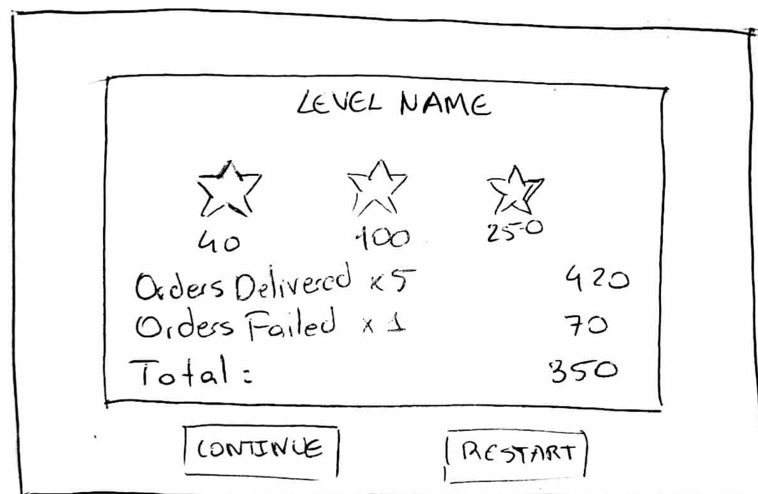

Figure 3.26: Concept Of Score Panel

3.7 Level Design

The next level is level 9, whose topic is Level Design (Rogers, 2010, p. 197). This chapter covers the most common types and techniques of level design, however, the suggestions in this chapter are intended for games of other types. In the case of Overforged, the levels are reduced in size, fitting completely into the camera's field of view, so the *Level Design* process of this game is simpler.
Level Design is a complex and extremely important area in the development of most video game genres. In *Overforged*, the **main concern for any of the levels** to be designed is to decide **which objects will be present in the level**, **where they will be placed**,

and **which obstacles may disturb the player while he forges**. To determine the objects needed, we need to take into account the steps of the forging process.

To forge a blade, the player must get the correct metal and put it in the furnace, wait for it to get hot, take it to the anvil, shape it and cool it. Figure 3.27 illustrates the process of forging a blade.

Figure 3.27: Process of forging a blade

To prepare the handle, the player must get the correct wood, place it on the saw and cut it into the desired shape. Figure 3.28 shows the process of making a handle.

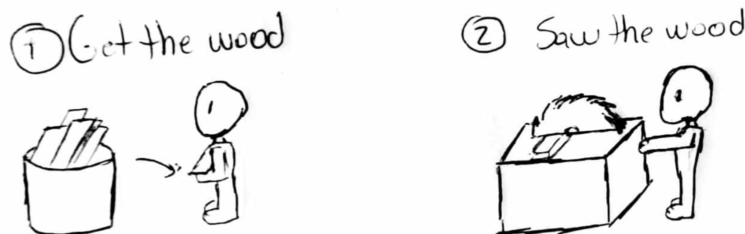

Figure 3.28: Process of making a handle

Finally, to make a simple blade, the player has to merge the blade and the handle. To merge them, one of the objects must be placed on a table, and the other must be in hand. Figure 3.29 illustrates how to merge the blade and the handle.

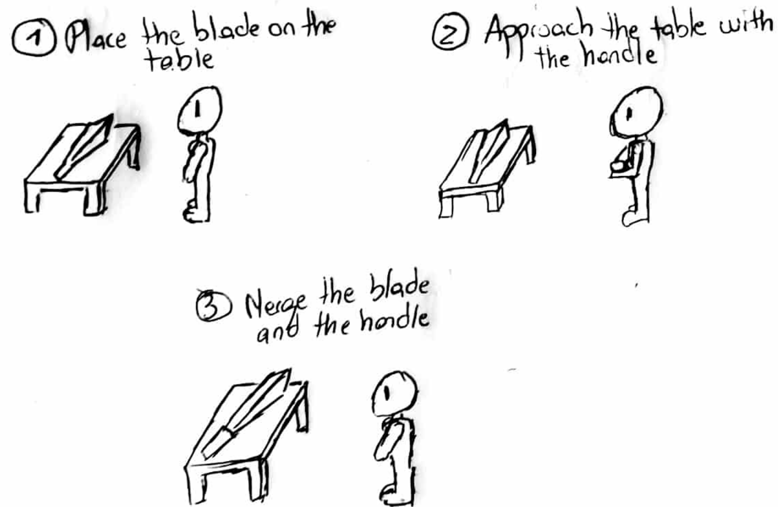

Figure 3.29: How to merge the blade and the handle

A guard or a string can also be added to weapons by placing the weapon on the table and having the guard/string in hand. A guard is forged just like the blade, using gold instead of iron, and once forged it can be added to the weapon. The string is a material that, once obtained, can be added. Figure 3.30 shows the results of merging the katana with the guard and the Viking axe with the string.

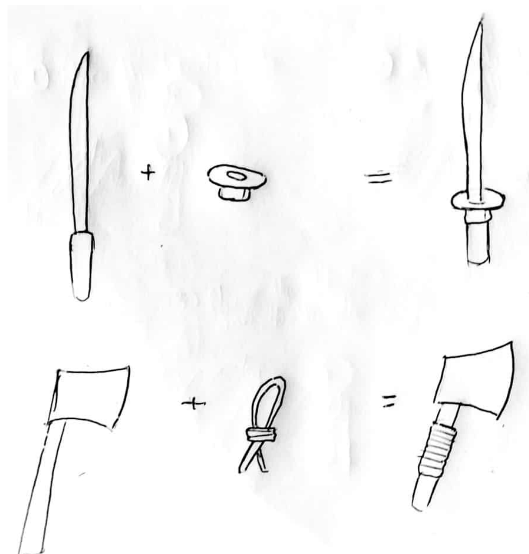

Figure 3.30: Results of merging the katana with the guard and the viking axe with the string

Finally, the weapon must be delivered to receive the points.

Given the aforementioned possibilities, the following objects can exist on a level:

- Object to spawn the iron;
- Object to spawn the gold;
- Object to spawn the string;

- Object to spawn the wood;
- Furnace to heat the metals;
- Anvil to shape the metals;
- Saw to cut the woods;
- Object for cooling metals;
- Table to join the components;

The player should be able to focus on several orders at the same time, otherwise, the level can be boring and slow-paced. To make the player aware of what to deliver, a set of 4 panels is placed in the upper corner of the screen, each of which has the object to be forged in the center of the panel, the necessary materials at the bottom of the panel and a slider with the time remaining to complete the order, as described in section 3.6.

Whenever **one of these orders is completed**, the **player earns X points**, the completed order disappears from the screen and a new order is placed on the screen. When an **order runs out of time**, the same thing happens, but instead of gaining X points, **the player loses X points**. This cycle continues until the level time runs out.

The type of **objects placed** on the level **varies with the type of orders** the level needs. As an example, in a level where the player has to produce katanas, where it is necessary to prepare two types of metals and only one type of wood, it makes sense that there are 2 anvils, at least 2 furnaces, and only 1 saw. After deciding on a good placement for the objects essential to the forging process, it is necessary to place balconies in the scenario, which have two main uses:

- Allow player to place objects;
- Be an obstacle to player movement.

Finally, the topology of the level's scenario is also an important obstacle to the player. For instance, putting a river in the scenario, we force the player to go around another path and cross a bridge.

The levels of *Overforged* will be implemented taking all these points into account. Figure 3.31 illustrates a concept of a level.

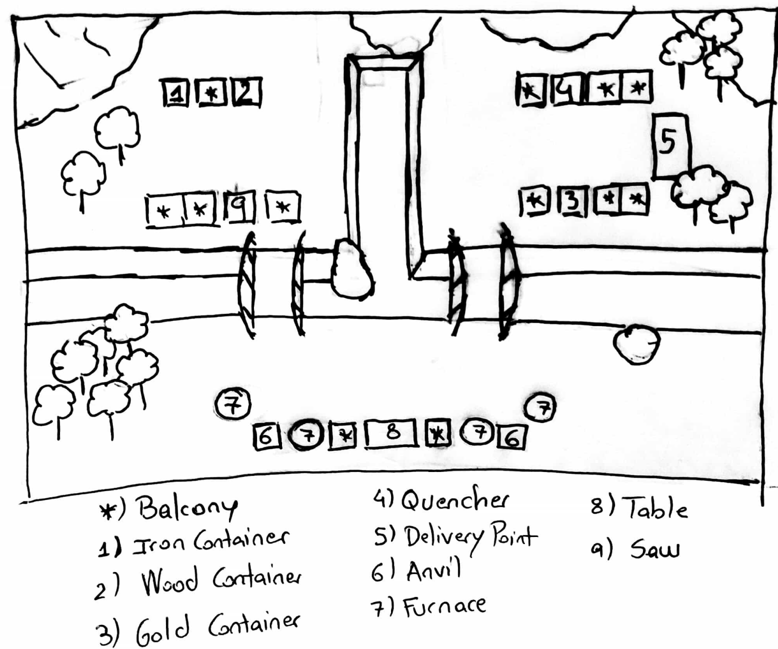

Figure 3.31: Concept Of Level

3.8 Mechanics

Levels 10 and 11 are about combat and enemies, which are elements absent from Overforged, making level 12, which is about mechanics, the next target for analysis. A game without mechanics is not a game, it's empty and boring. Mechanics are, according to Scott Rogers, objects that create gameplay when the player interacts with them (Rogers, 2010, p. 331). In *Overforged*, we have the following list of mechanics:

- Grab and drop objects;
- Throw objects;
- Take objects from containers;
- Use balconies;
- Use the furnace;
- Use the anvil and the saw;
- Use the quencher;
- Use the table;
- Deliver objects;
- Change character control.
- Kill and Respawn a Character

3.8.1 Grab And Drop Objects

In the levels, the **player must** be able to **grab**, **carry** and **drop objects**. The player can grab an object when he is close to that object, whether it is lying on the ground or resting on a balcony. When the player grabs the object, that object is transported to the player's hands and remains there until the player drops it. Figure 3.32 illustrates the concept of the player grabbing an object from the ground.

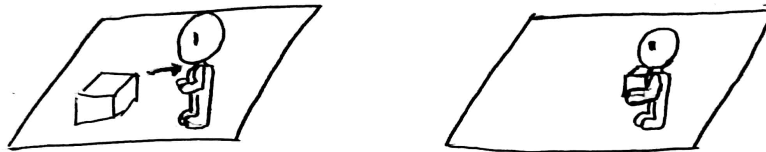

Figure 3.32: Concept Of The Player Grabbing An Object From The Ground

When dropping the object, if a counter is in front of the player, that object will be placed on the balcony, otherwise, it will be dropped to the ground. The figure 3.33 represents the two possible outcomes when the player drops an object.

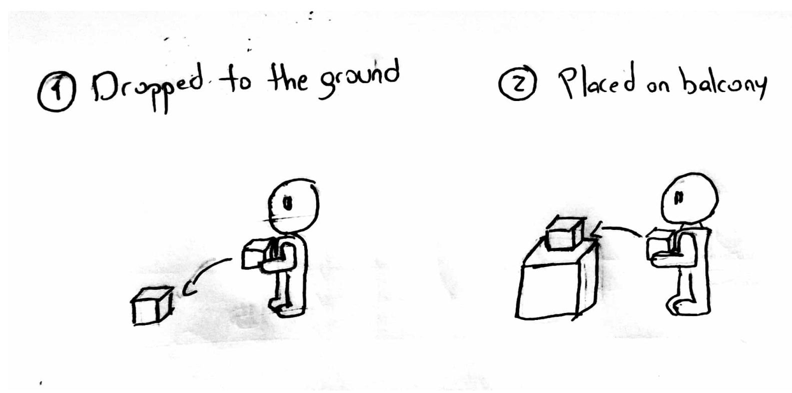

Figure 3.33: The two possible outcomes when the player drops an object.

3.8.2 Throw Objects

When a **player carries an object**, instead of dropping it, **the player can throw it**. When throwing an object, this object will travel in a uniform rectilinear motion until it hits the ground. If the object hits a balcony before falling, that object will stay on the balcony as shown in figure 3.34:

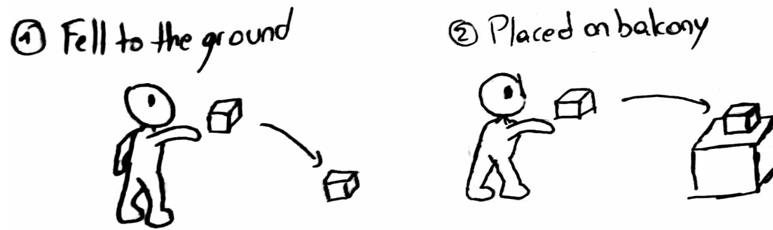

Figure 3.34: The two possible outcomes when the player throws an object.

3.8.3 Take Objects From Containers

When the **player approaches a material container**, the player **can withdraw a material** instance from the container. This removed material will be immediately transported to the player's arms so that he can carry it. There is no limit to the number of instances of material the player can retrieve from the container. Figure 3.35 shows the player taking the material from the container.

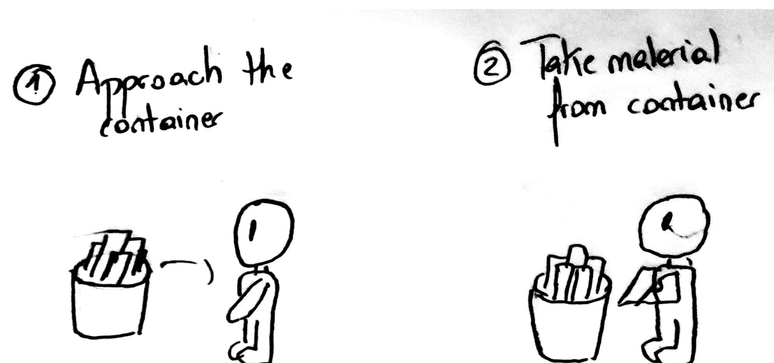

Figure 3.35: Concept of the player taking an material from the container.

3.8.4 Use Balconies

When the **player carries an object and approaches a balcony**, the player can **land that object on the balcony**. In the common balconies, any object can be placed, as long as it is free, as only one object can be placed on each balcony. There are, however, some types of balconies where objects can only be placed under specific conditions, such as:

- **Furnace** - balcony where you can only place metals in their base state;
- **Anvil** - balcony where you can only place metals in their heated state;
- **Saw** - balcony where you can only place wood in its base state.

In the same way that the player puts down the object, he can also, in most situations, remove it. **When an object is on a common balcony or in the furnace, it can be removed at any time**. **When an object is in the anvil or saw, that object**

cannot be removed until it is fully hammered/sawn. Figure 3.36 illustrates the player placing and taking an object from the balcony.

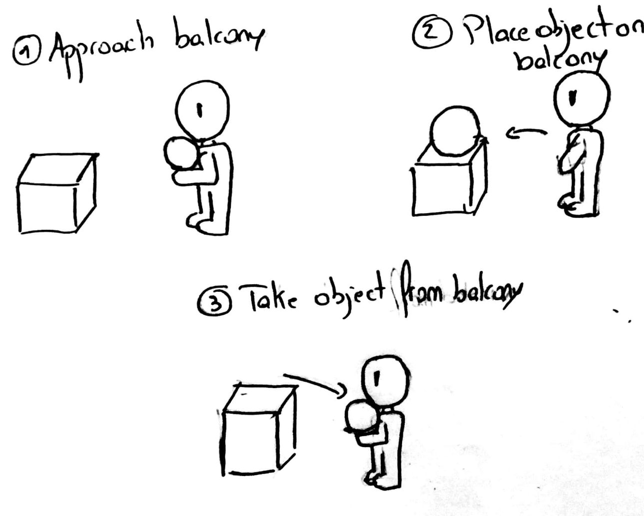

Figure 3.36: Concept of the player placing and taking an object from the balcony

3.8.5 Use The Furnace

When the player carries a metal in its base state, that object can be placed in the furnace. When the object is placed in the furnace the heating process begins and a progress bar appears. When the progress bar reaches 100% the metal switches to the heated state and a new progress bar starts to fill. The player must remove the metal before the bar fills, or it will burn. When the metal is burned, it becomes useless. Figure 3.37 represents the player heating metal on the furnace.

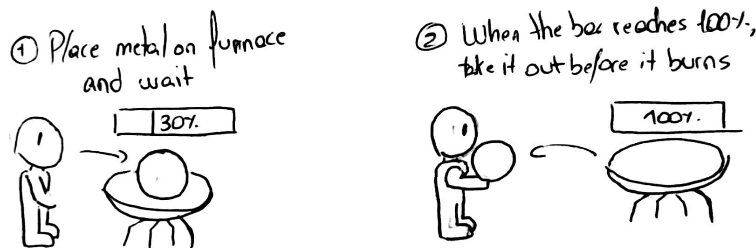

Figure 3.37: Concept of the player heating a metal on the furnace.

3.8.6 Use The Anvil and The Saw

When a player carries a heated metal, it can be placed in the anvil. Once the metal is placed in the anvil, it cannot be removed until the player starts and completes the hammering process. The player must hold a certain input to maintain the hammering action. As the action progresses, a progress bar appears, and the hammering animation loops. When the bar reaches 100%, the metal turns into a blade in its heated state. The blade can now be removed and loaded.

The saw works in the same way, however, instead of heated metal, the player must place an instance of wood in the saw and initiate the action. The process will be identical only varying the animation played. When the progress is 100%, the wood will become a handle. Figure 3.38 shows the player shaping metal on the anvil.

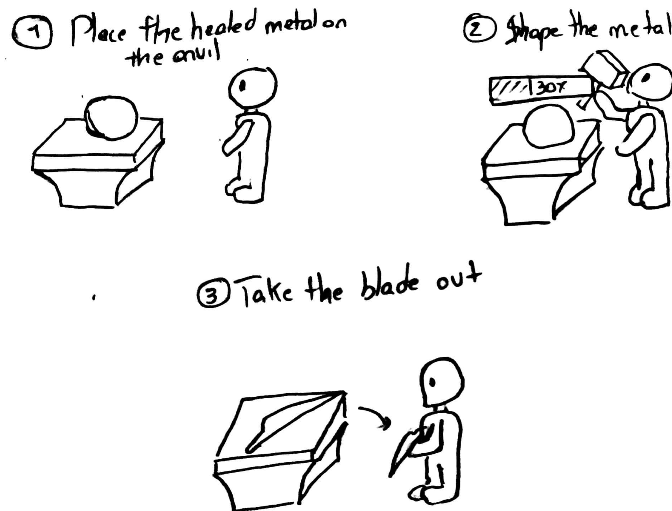

Figure 3.38: Concept of the player shaping metal on the anvil.

3.8.7 Use The Quencher

To cool a heated blade, the player must carry it to the quencher and interact with the quencher. An animation of the player dipping the blade into water will play and the blade will change to its final state. The quenching action cannot be stopped. Figure 3.41 illustrates the player quenching a blade.

Figure 3.39: Concept of the player quenching a blade.

3.8.8 Use The Table

To join two objects, the player must carry one of the objects to the table and place that object on it. Then, carrying the other object, the player must approach the table and interact and an action that cannot be interrupted will be initiated. The action consists of playing an animation, at the end of which the two objects will be merged. Figure 3.40 represents the player merging two objects on the table.

Figure 3.40: Concept of the player merging two objects on the table.

3.8.9 Deliver Objects

To deliver an object, the player must carry that object to the delivery point. When delivering the object, it will be destroyed and if the delivered object matches one of the requested orders, the player will earn points. Figure 3.41 shows the player quenching a blade.

Figure 3.41: Concept of the player quenching a blade.

3.8.10 Change Character Control

***Overforged* is a game that can be played by either one player or two players**. In case the player is **playing solo**, it is essential that the player can **control both characters** to be able to perform as many tasks simultaneously. In case two players are playing, there are levels where the characters are separated in different positions, and in this case, players may prefer if their characters were in the other position. For both situations described, there is the mechanic of changing the control of the characters.

When the player plays alone and tries to change control of the character, this change is immediate. If the previously controlled character was using an anvil, saw, quencher, or table, that character's action will continue to run until it ends. Figure 3.42 illustrates what happens when the player tries to change the controlled character when playing solo.

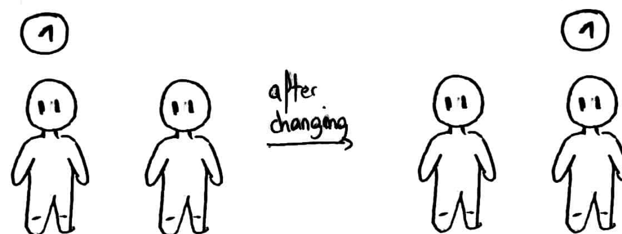

Figure 3.42: What happens when the player tries to change the controlled character when playing solo.

When two players are playing, one player must express the intention to exchange, and the other player must do the same within a 3-second window, and if he does, the exchange takes place. Figure 3.43 shows what happens when the player tries to change the controlled character when playing in co-op.

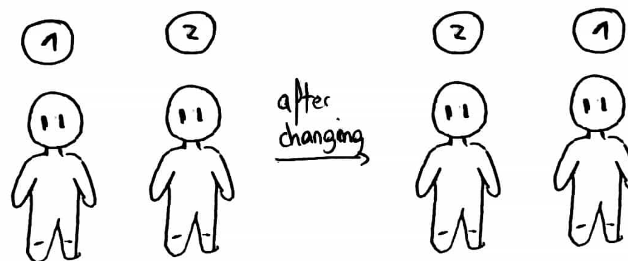

Figure 3.43: What happens when the player tries to change the controlled character when playing in co-op.

3.8.11 Kill and Respawn a Character

During the levels, **characters can die**, be run over, or drown, for example. In this case, the characters must play a death animation, and at the end of the animation, disappear. After disappearing, they should **respawn at their spawn point after 3 seconds**. Figure 3.44 shows what happens when the player is run over.

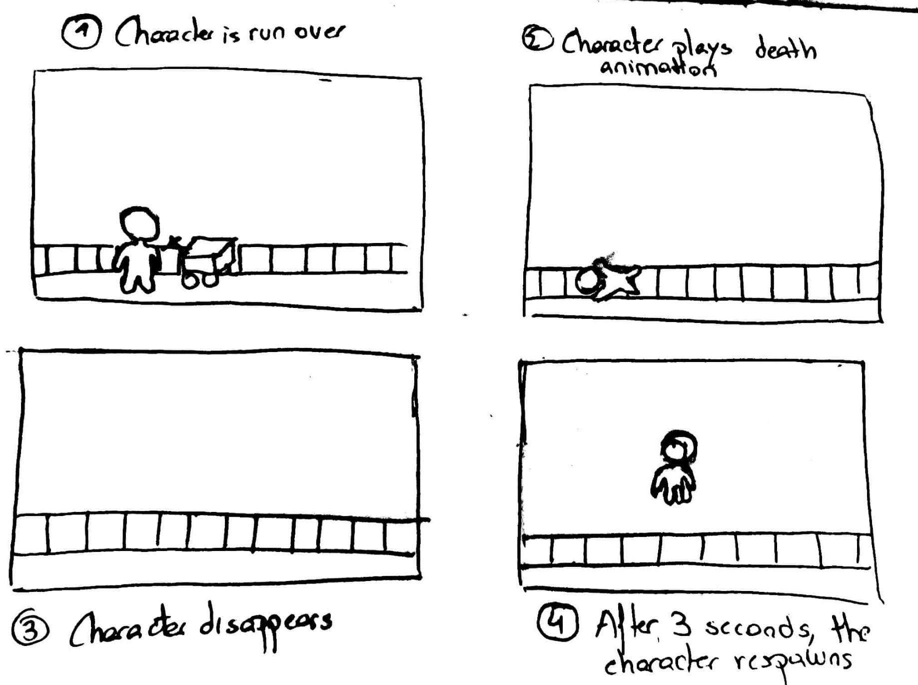

Figure 3.44: What happens when the player is run over.

3.9 Multiplayer

Level 13 is about power-ups, which are not present on *Overforged*, so following the levels of the book, the next level is level 14 which is about multiplayer. According to Scott Rogers, multiplayer can be done through 3 connection types (Rogers, 2010, p. 382):

- **Head-to-head** - two or more players play in the same game system;
- **Network/Peer-to-peer** - two or more players play on different machines connect via LAN or Network;
- **Client-server LAN** - two or more players play on different machines connected to a computer.

In addition to the connection type, there are 3 multiplayer game styles (Rogers, 2010, p. 382):

- **Competitive** - players work against each other;
- **Cooperative** - players work together towards the same goal;
- **Conjugate** - players work together towards the same goal, but also compete against each other while pursuing that goal.

***Overforged* is a local co-op game**, that is, it is a game **with a head-to-head connection** type and **cooperative style**. In the future, we might add a network connection system and competitive modes.

In *Overforged*, multiplayer makes getting 3 stars more accessible than singleplayer and makes for much more fun gameplay. The game is limited to two players, with the possibility of being expanded in the future to be played by 4 players, both locally and over the network.

There won't be any kind of direct interaction between the players, and they will all be able to do the same set of actions.

3.10 Music and Sounds

The next level is level 15, whose topic is Music. Scott Rogers suggests that instead of each level having associated music, it would be more dynamic to use a set of sounds that reflect the mood of the game (Rogers, 2010, p. 393). However, despite the suggestion, the different levels of *Overforged* take place in different environments, and each level will have a song associated with the theme of the level. **Every sound used in the game is royalty-free**.

Next, the author lists a set of sound effects that can be framed in a game (Rogers, 2010, p. 400), from the sound of movement to the sound of victory. In *Overforged*, the following set of sounds will be implemented:

- **Dash** - sound played when player dash;
- **Throw object** - sound played when the player throws an object;
- **Saw Wood** - sound played when the player saws wood;
- **Hammer Metal** - sound played when player hammers metal;
- **Quench Blade** - sound played when player cools metal;
- **Merge Items** - sound played when the player merges two objects on the table;
- **Fall in Water** - sound played when the player falls into the water;
- **Ran Over** - sound played when the player is run over.

All the sounds mentioned are reproduced **in 2D space**, that is, the sound will always have the **same volume regardless of the position** of the Audio Source and the distance from it.

Chapter 4

The Development of Overforged

4.1 Setting Up The Unity Project

The first step in the development of *Overforged* is the creation of the Unity project. As a template, we choose the *3D Universal Render Pipeline*. Figure 4.1 shows the project creation screen as well as the chosen template:

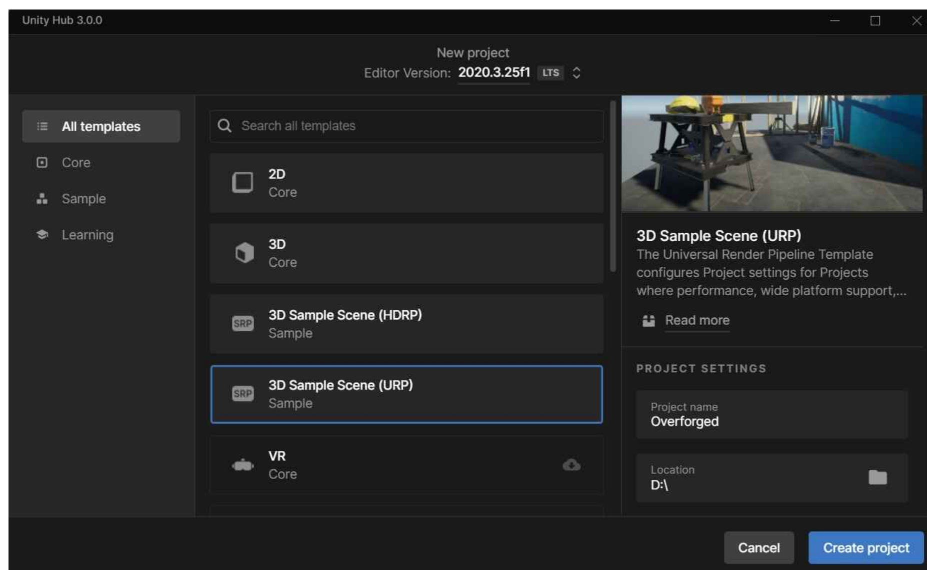

Figure 4.1: Project creation screen and the chosen template

Unity offers 3 graphical rendering pipelines, the Universal Render Pipeline (URP), the High Definition Render Pipeline (HDRP), and the *Built-In Render Pipeline*.

We chose the URP (Unity, 2022f) as a render pipeline because the features offered by the HDRP (Unity, 2022b) are not needed, the latter would only bring performance costs and very few graphical benefits. Another reason for choosing the URP is **because it is more customizable** than the built-in pipeline and **allows the development of a project with distinct and specific graphics**.

4.1.1 Setting Up The Input System

With the project created, we need to setup the interaction between the game and the user, in order to achieve this we need to setup an input system. **The Unity's new Input System was the choice**, because it is Unity's new bet, which manages to offer **a system that is adaptable to different platforms**. Figure 4.2 shows the new Input System package.

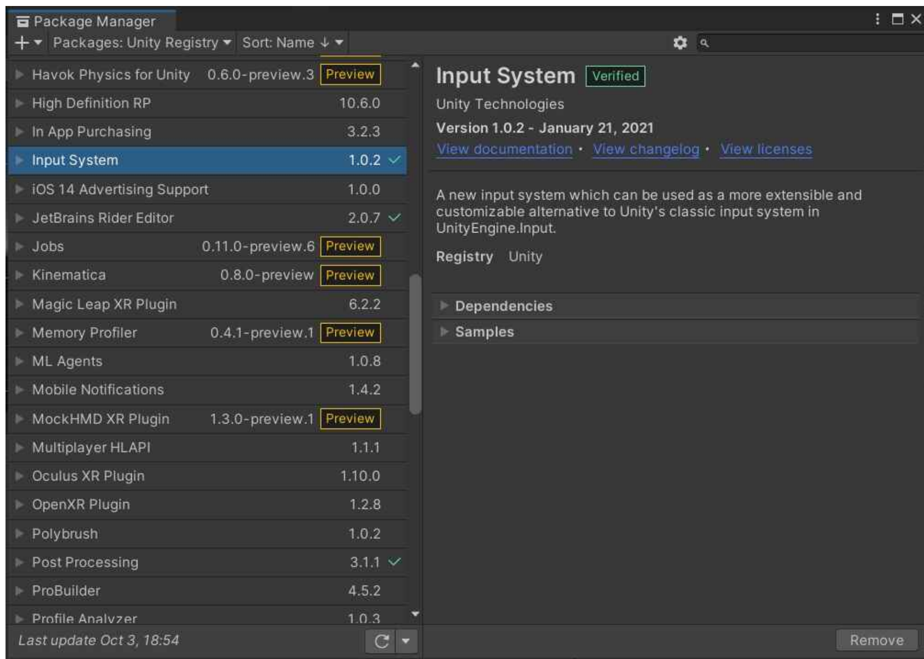

Figure 4.2: The New Input System Package

To set up the inputs, we must create an asset called *InputActions*, this asset will store all possible actions of the game, and it's also responsible for sending out events that allow us to know the action was made.

Figure 4.3 shows the representation of the *InputActions* asset.

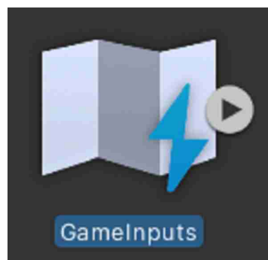

Figure 4.3: Representation of the InputActions asset

In Unity's input system, we can create action maps that are a method of organizing Inputs into categories, within these maps, there are sets of actions that the player can perform, and each action is composed of a set of buttons that trigger the action.

As an example, an action map might be the *Gameplay* map that contains the inputs that can be triggered during the playable levels of the game.

Figure 4.4 represents the Gameplay map created for the game.

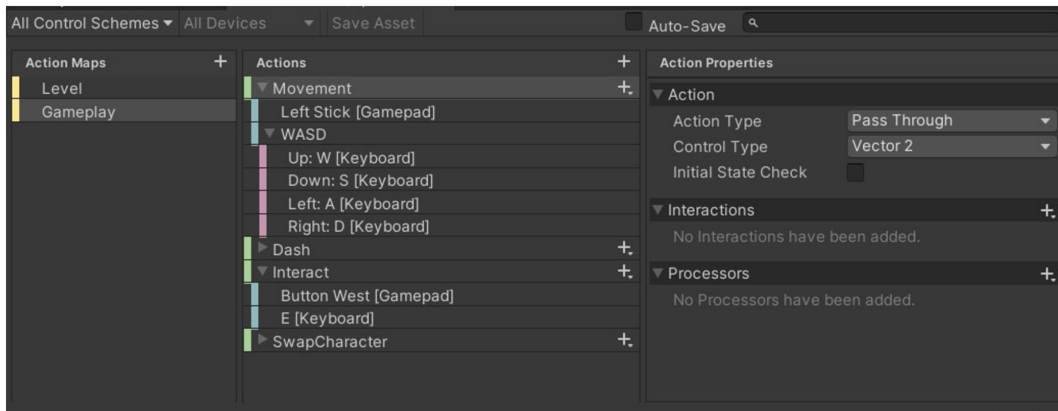

Figure 4.4: Gameplay Input Map in the InputActions Asset

The input system offers a wide range of action types and control types, but for *Overforged*, only two types of actions will be used:

- **Pass-Through**: Used for floating values such as vectors. In this project, this type of action will always be used with a Vector2 return type;
- **Button**: Used for binary values;

Inputs such as ”Movement”, which can take values between -1 and 1, will have the action type ”Pass Through”. Inputs such as ”Dash”, ”Interact”, and ”SwapCharacter” will be of the ”Button” type, as they can only assume two states: true (Clicked) and false (Not Clicked).

Actions that return a 2D vector can be associated with a specific key (such as the Left Stick of a Gamepad) or with a 2D key composition, where each key corresponds to a part of the composition. For the motion input, we used a key for the Gamepad and a 2D composition for the keyboard, as shown above in figure 4.4

Now, to receive the result of these inputs, we have to link a set of functions to them, this link can be done in any script that has an instance of ”GameInputs”. The following code snippet shows how to create an instance of the GameInputs and listen to the Dash input action:

```
gameInputs = new GameInputs(); //Create an instance of the GameInputs

gameInputs.Gameplay.Dash.performed += context => OnDashInput(ctx.control.
    device, ctx.ReadValue<bool>()); //Listen to dash input press
gameInputs.Gameplay.Dash.canceled += context => OnDashInput(ctx.control.
    device, ctx.ReadValue<bool>()); //Listen to dash input release

void OnDashInput(InputDevice device, bool value)
{
    //Do something
}
```

Code Snippet 4.1: Creating an instance of GameInputs and listening to the Dash input action

Whenever an input is triggered, a set of information is sent, including the device that acted and the result of the action. These two values are a must in our game, as **we need to know which device acted** since each player has a device bound to it, and the value of that input. If we don't check which device acted, then all players would receive all the inputs from all the devices.

For 2D Vector type inputs, these inputs are never canceled, because even when the player is not performing any action, these inputs return the value (0, 0), so we just send the value, stored in "Context", to the function as shown in the snippet below:

```
gameInputs.Gameplay.Movement.performed += ctx => OnMovementInput(ctx.
    control.device, ctx.ReadValue<Vector2>());


void OnMovementInput(InputDevice device, Vector2 value)    {
    uiNavigateInput = value;
}
```

Code Snippet 4.2: Binding a Vector2D action to a function

This is how we will listen to any input we need in any script.

4.1.2 Setting Up The Devices Logic

As this game is a local cooperation game, which can be played with both keyboard and gamepad, we **need a system that tries to detect the connected devices and distributes them to the players**. To this end, a script called "DevicesManager" was created, which is always active while the game process is open, in this script, a list of all connected devices and the devices assigned to each player is kept.

To detect whether a device was connected or disconnected, the "onDeviceChange" callback of Unity's InputSystem is used. This callback is invoked when the state of a device is changed, returning the device in question and its new state. The code snippet below shows how to use the "onDeviceChange" callback.

```
InputSystem.onDeviceChange += (device, change) =>
{
    switch (change)
    {
        case InputDeviceChange.Added:
            RefreshDevices();
            break;
        case InputDeviceChange.Removed:
            RefreshDevices();
            break;
        ...
    }
};
```

Code Snippet 4.3: Listenning to the onDeviceChange callback

When the game starts, or whenever a device is connected/disconnected, the devices are refreshed and their assignment is made. The refresh process consists of getting all gamepads connected and assigning them to players in order of connection. Player 1 has the option to play with a gamepad or keyboard. If the player chooses to play with the gamepad, the first gamepad to be connected will be assigned to player 1, otherwise, it will be assigned to player 2.

The figure 4.5 represents the devices logic used in the game.

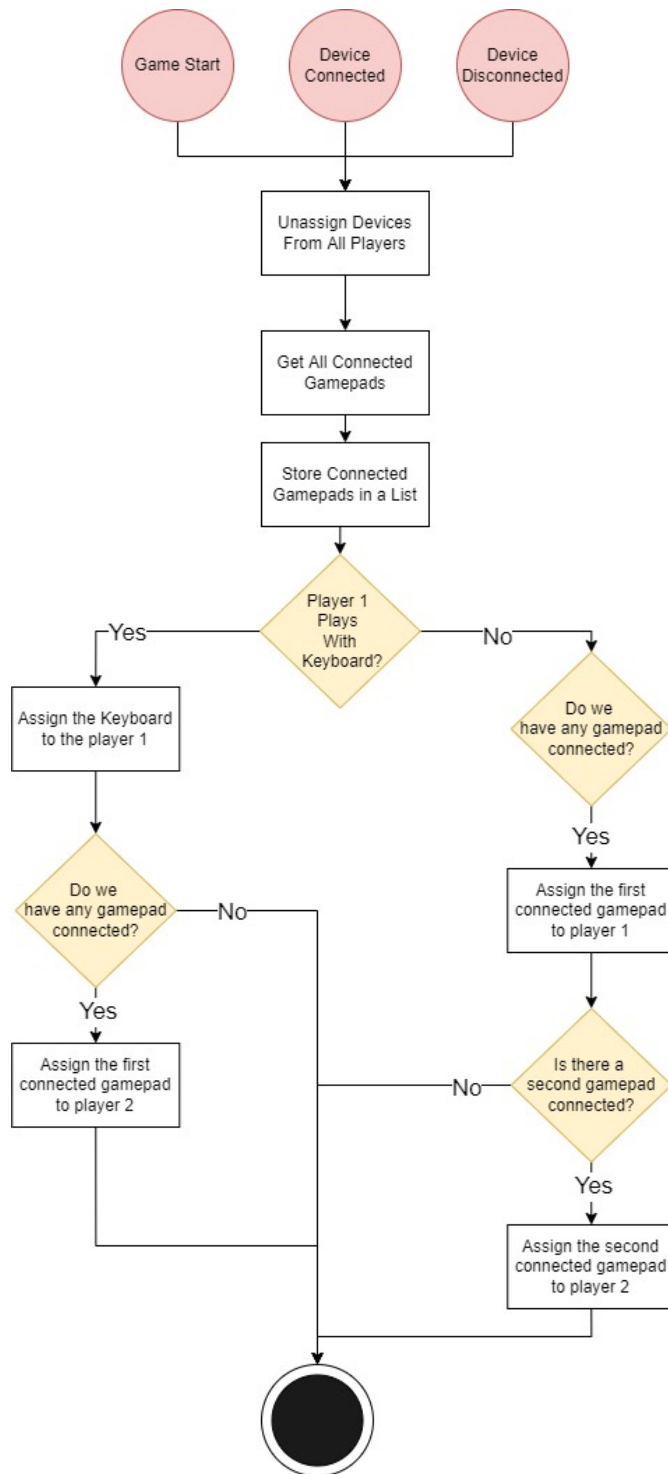

Figure 4.5: Devices Logic Used In The Game

4.2 Game Bases

The next stage is the development of some base systems that will facilitate all the remaining stages of development.

4.2.1 Player Base

First, we have to develop a base for our characters. To start creating and implementing the first features of the game, it is necessary to have at least a base for the player.

To represent the playable characters of *Overforged*, the package ”POLYGON MINI - Fantasy Characters Pack” from Synty Studios was acquired. This pack can be found on the Synty Store website (Synty, 2022a) and it offers a set of 60 characters with 6 possible color sets that are ready to use. The pack also offers 28 props of different types, from weapons to scenery components. The figure 4.6 represents the ”POLYGON MINI - Fantasy Characters Pack” from Synty Studios, used for the characters.

Figure 4.6: POLYGON MINI - Fantasy Characters Pack, Synty Studios, from Synty Store (Synty, 2022a)

The imported character pack has a player prefab, which will be used as the starting point for the player base. This prefab has an armature made of several bones used to animate the character and a set of objects corresponding to the different models. To keep the project more organized, the models and the armature were placed as children of an object called ”Models”, as shown in figure 4.7.

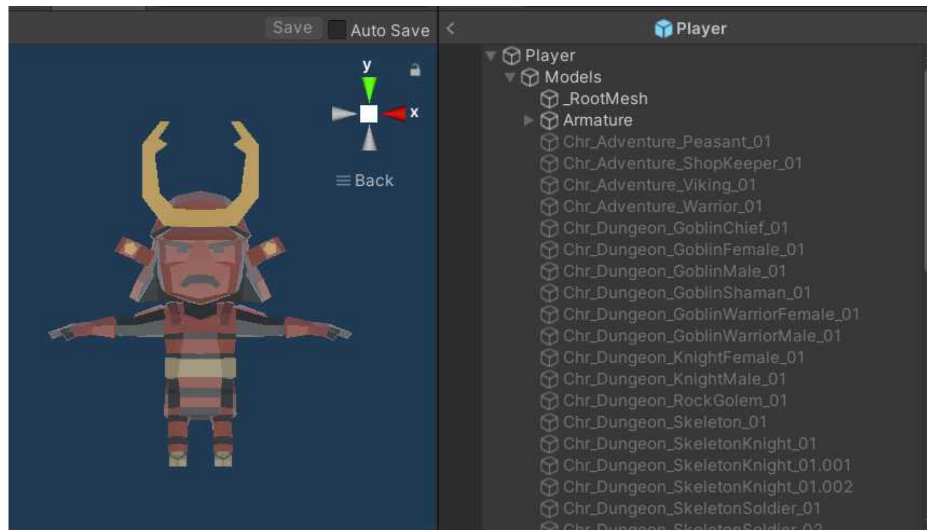

Figure 4.7: Fantasy Characters Prefab

With this prefab as a starting point, we start the development of the player base by creating a **sphere above the head of the character**, **indicating the character is being controlled by the player**. The sphere will be colored in red if the character is being controlled by player 1 and in blue if the character is being controlled by the player 2. We also add the following components:

- **Animator** - component that sends the animations data to the armature bones;
- **Rigidbody** - component to simulate the physical behavior of a rigid body;
- **Capsule Collider** - component that allows this object to collide with others;
- **PlayerInputs** - script where the input values of the player are stored.
- **PlayerModel** - script where there are references to all the bones that make up the armature, as well as the models associated with the player.

In the Rigidbody component, we check the ”FreezeRotationX”, ”FreezeRotationY” and ”FreezeRotationZ” checkboxes as we want to manually control the player’s rotation.

The ”PlayerModel” script makes it easy to switch models because we have references to every model. **If we want to switch to another model, we just deactivate the current model and activate the new one**. This script will be extremely useful for programming future interactions, where references to the bones of the armature will be needed.

The last step needed to complete the player base is to add animations to the animator. **An animator is made up of states, and each state plays an animation**. The transition between these states can be done in the animator or manually through code, in *Overforged* transitions will be done manually. The states are further divided into 2 categories:

- **State** - simple state, with an associated animation;

- **Blend Tree** - complex state, with n associated animations. These animations can have k parameters associated, and the animations will be mixed according to these parameters.

As the player's first animation state, a Blend Tree called Locomotion was created, composed of 3 animations: Idle, Walk and Jog. The tree result will be controlled by 2 parameters, a horizontal speed, and a vertical speed. Figure 4.8 shows the Blend Tree Locomotion. In the figure we can see that when the vertical speed (corresponding to the Y-axis) is 0, the animation to play will be "Idle", when it is 0.5, the animation will be "Walk" and when it is 1 the animation to play will be "Jog". Any value in between will play a mix of two animations. Although the horizontal speed parameter is not used at the moment, it may be useful in the future for strafing animations.

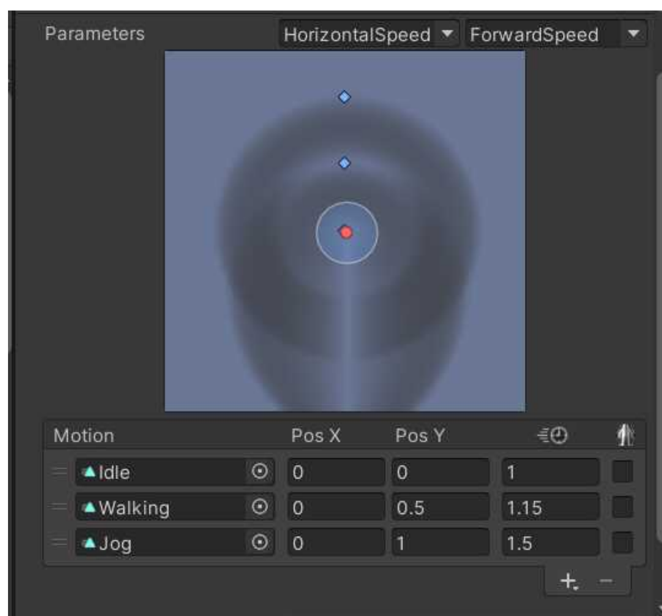

Figure 4.8: Animator Locomotion State

The base is now ready to be expanded when needed.

4.2.2 Level Management

Now that we have a player base to work with, the next step is to deal with the logic in each level of the game. These levels are divided into 3 categories:

- **Main Menu Level:** - Level corresponding to the game's main menu;
- **World Map Level:** - Level corresponding to the game's world map;
- **Smithing Level:** - Playable game levels.

All these level categories have different functionalities and common functionalities, so it makes sense that **they all derive from the same base**, this base is called a "LevelManager".

In the "LevelManager", we have:

- **Reference to two players:** - so that any script can access them;
- **AudioSource for the Music** - Audio source in which the music of the level will be played;
- **Level Music Audio Clip** - audio clip corresponding to the level's music;
- **Function for switching character control:** - function so that players can switch the controlled character;

All levels that derive from this class will have access to player references and the trade function.

The swap function is called when one of the two players presses the input action "SwapCharacter". The player tells the LevelManager that he wants to switch control of the player, and the LevelManager arranges for the switch if possible.

If there is only one player connected the exchange is immediate, however, if there are two players connected, if player X presses the exchange button, player Y has 3 seconds to press the exchange button as well or the exchange will not occur. The swap consists of swapping the input devices of the two players.

Along with the LevelManager, the LevelInputs script was also created, in this script an instance of GameInputs is created as mentioned in the 4.1.1 section, and the inputs received in this script are used for UI navigation, except in the case of "World Map Level", where they are also used to move players.

Once the level starts, the audio clip containing the level's music is played in the AudioSource, which is done using the Play(AudioClip) function of the AudioSource class. This concludes the development of the Level Base.

4.2.3 UI Base

The development of UI can be quite complicated and sometimes repetitive. To ease the whole process and avoid repetition and redundancy, a good foundation for UI is essential. **In Unity, the user interface is drawn on a Canvas (Unity, 2022a)**, this is an area where graphical elements are inserted. The order in which elements are inserted into a Canvas is relevant, as these elements are drawn from first to last, which means that if two elements overlap, the last one will be rendered on top.

As stated in Unity manual (Unity, 2022a), Canvas can be rendered in 3 different ways:

- **Screen Space - Overlay** - in this mode, the Canvas is rendered over all elements of the scene;
- **Screen Space - Camera** - in this mode, the Canvas is also rendered over all elements of the scene according to a specific camera. This canvas is placed a certain distance from the camera;
- **World Space** - in this mode, the Canvas is rendered like any other scene element, having a position, rotation and scale.

In *Overforged*, **we will always use the "Screen Space - Overlay"** mode as we want to render the Canvas above the scene elements. **The only exception is the UI of the Main Menu**, this is blended and positioned together with the rest of the scene elements and is therefore **rendered in "World Space" mode**.

In the UI of the game in question, there will be the following types of elements:

- **Text** - used to write text on the screen;
- **Image** - used to render images on the screen;
- **Button** - built from Text and Image elements;
- **Slider** - built from Image elements.

4.2.3.1 UI Slider

Although Unity offers components that perform the function of Slider and Button, we chose to build these elements manually to have more control over them. To build the Slider we use a set of 3 images, a background, a fill, and a control. The background image will be visible behind the fill and the control image is a transparent image, which acts as a mask for the fill.

In Unity, a Mask is an image, which renders the child elements only in the area where it is visible. In the case of the slider, if the control image has a width of 100 units, then the fill image will have a width less than or equal to 100 units, but never more than 100 units, as this is the maximum size of the Mask. Therefore, we will use the control image to show and hide the fill image according to the value of the Slider.

To make the control image stretch according to the value of the slider, this image needs to be marked as a "Filled" image, and with the "Horizontal" method, as we want the Slider to work horizontally. If we want the slider to work vertically, it is simpler to rotate this Slider 90º than to change any of the settings. In the "Filled" image type, we have a value called "Fill Amount", which varies between 0 and 1, when the value is zero the image is completely hidden in the left corner and when the value is at 1 the image fills the slider.

Figure 4.9 shows the slider when the image has a "Fill Amount" of 0.7.

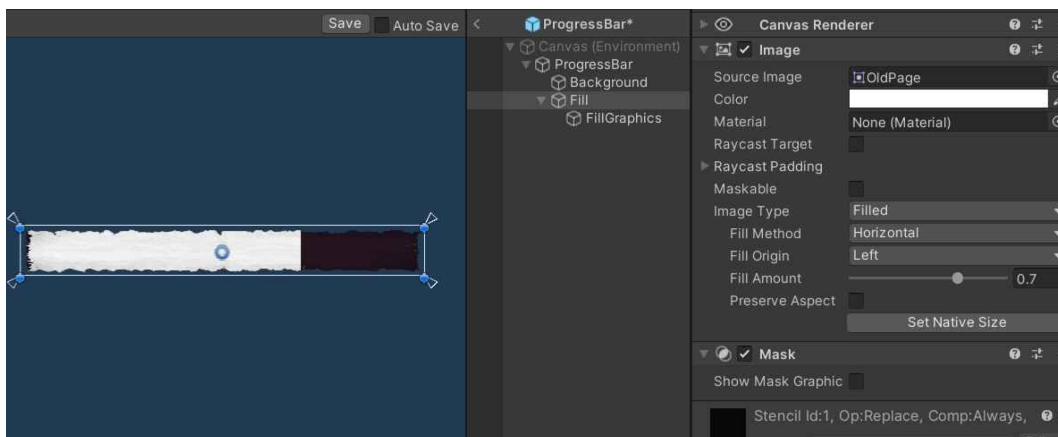

Figure 4.9: Slider UI Component

To this slider, we will associate a script called ”ProgressBar”, in this script we have the function SetValue(float newValue), used to control the ”FillAmount” of the control image, so that we can update the slider later.

4.2.3.2 UI Button

A button is an object that is divided into 2 states, the ”Normal” state, and the ”Hovered” state. The content of these two states will be diverse and irrelevant in terms of functionality, being only important visually.

Figure 4.10 shows the ”Normal” and ”Hovered” states of the button. In the button represented in the image, each state has an image for the background, an image for the border, and a text, the latter being identical in content in both states. The purpose of this arrangement is to be able to create a set of visual elements and switch between them when the player selects/unselects the button.

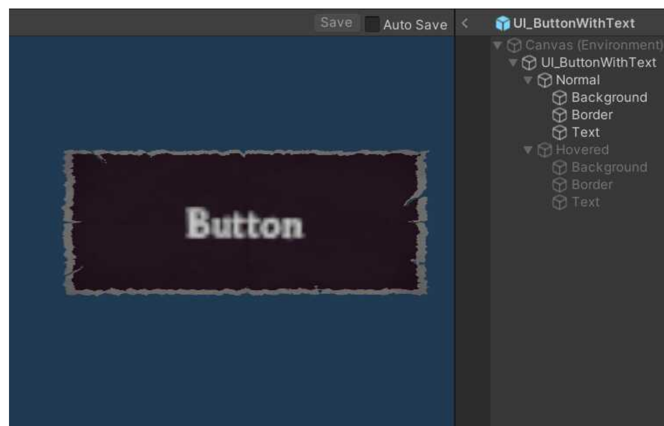

Figure 4.10: Slider UI Component

To implement the various functions of the button, the script *UI_Button* will be created and associated with the button, in this script, we will have a reference to the ”Normal” and ”Hovered” states of the button so that we can switch between them, and we will have the following events:

- **OnButtonClicked** - event that warns the subscribed methods that the button has been clicked;
- **OnButtonHovered** - event that warns subscripted methods that the cursor is over the button.

Events in Unity are typed as *delegates*, that is, types that represent references to methods with the same set of parameters (Microsoft, 2022b). Methods with the same signature as the event can subscribe to the event, and whenever the event is invoked, those methods are executed. For the button, the methods mentioned are:

- **ClickMe()** - function called when the button is clicked;
- **HoverMe()** - function called when the cursor is placed over the button.

To detect the mouse position over the button, the *EventTrigger* component is required. This component offers the possibility of listening and reacting to different events that influence the object to which it is associated, of these events we are interested in *PointerClick* and *PointerEnter* triggered when the mouse clicks the button and when the mouse is positioned over the button, respectively. Figure 4.11 represents the *EventTrigger* component placed on the button.

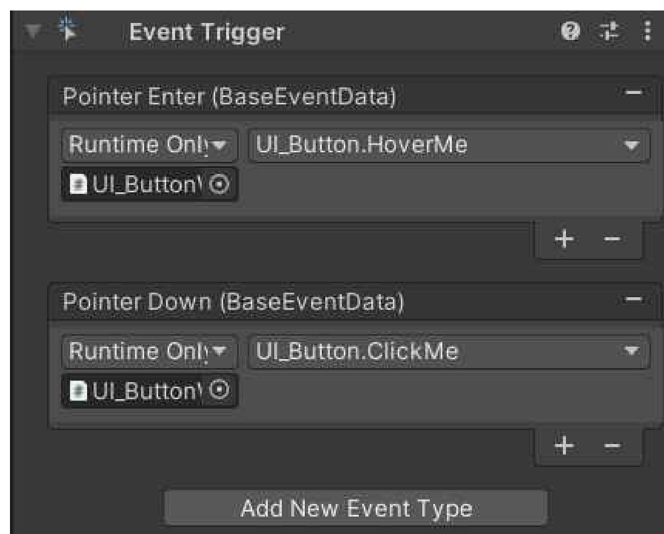

Figure 4.11: EventTrigger component of the button

As shown in figure 4.11, the events will call the *ClickMe()* and the *HoverMe()* functions so that they can call the *OnButtonClicked* and *OnButtonHovered* events. The figure 4.12 illustrates the logic used for events related to the button.

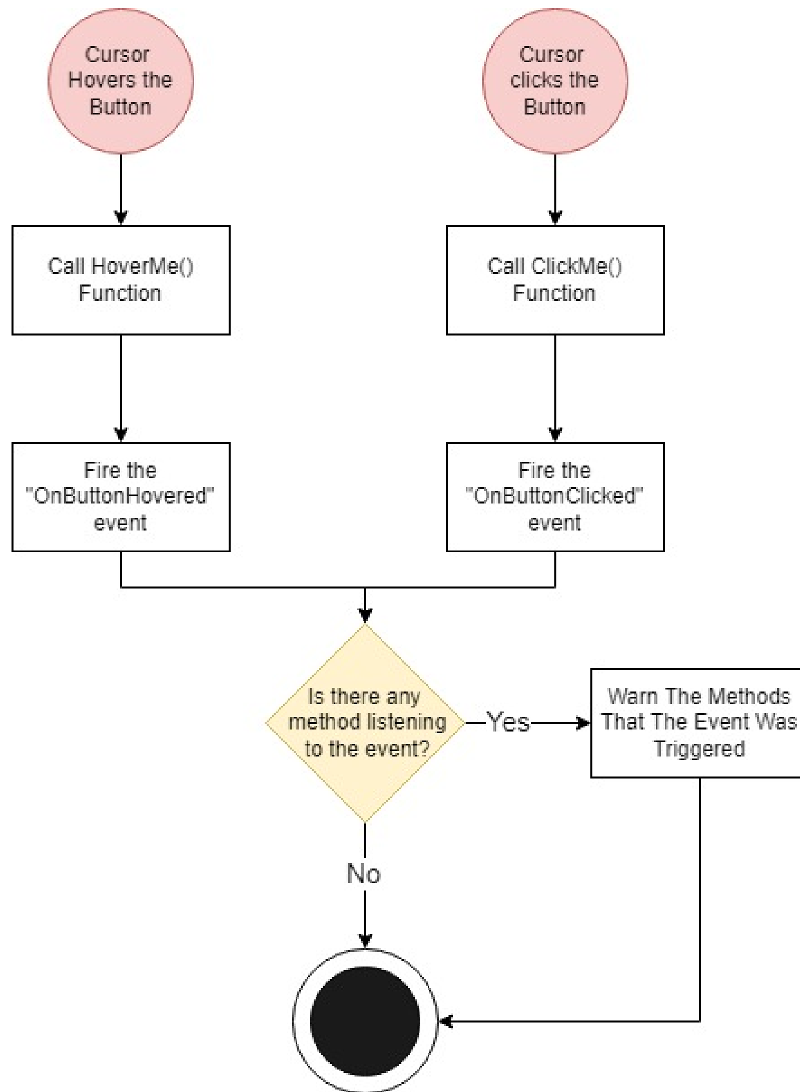

Figure 4.12: Button Events Logic

For the button, it remains now to create the keyboard navigation functionality. Unity offers built-in functionalities that allow you to configure navigation between the different buttons, but we chose to create our system to configure navigation with more freedom and precision. For keyboard navigation, a system of neighborhoods was created, in which each button has a list of neighbors, and is connected to that neighbor through a direction. The following code snippet shows the list of neighbors as well as the neighbor class.

```
UI_Neighbour[] listOfNeighbours; //list of neighbours

public class UI_Neighbour //Neighbour class
{
    public UI_Button neighbour; //Reference to the neighbour button
    public Vector2 direction; //Direction to the neighbour
}
```

Code Snippet 4.4: Class "Button Neighbor" used to configure the button neighborhood system.

The button is now ready to be implemented in the different menus of the game.

4.2.3.3 UI Menu

In *Overforged*, each menu is Canvas with the respective elements and all menus have common features and variables, such as:

- **DefaultHoveredButton** - reference to the button that must be activated once the menu is opened;
- **CurrentHoveredButton** - reference to the currently active button;
- **Open()** - function to open the menu, activating the Canvas of the menu;
- **Close()** - function to close the menu, deactivating Canvas of the menu;
- **OnConfirm()** - function called when "Confirm" input is triggered;
- **OnBack()** - function called when "Back" input is triggered;
- **OnNavigation(Vector2 direction)** - function called when "Navigation" input is used.

In a menu, every frame we check the navigation input, and if this is not zero, we will look in the list of neighbors (mentioned in section 4.2.3.2) of the currently selected button if there is a neighbor in that direction. If a neighbor is found then this will be the new active element, and we will change the state of the new element to "Hovered" and the state of the old one to "Normal". In the same way, at each frame, we check the input of "Confirm", which will work like a mouse click on the active button, and the input of "Back", which will go back to the previous Menu. In the 4.3 section, an example of a menu will be explored.

4.2.4 Game Settings

To let the player tweak and consult the game settings, a game needs a settings screen. To create the settings screen, first, we need to be able to tweak the settings in code, and for that, a script called *GameSettings* was created. This script, unlike other Unity scripts, does not derive from *MonoBehavior*, which means that it cannot be associated with a game object, it works just like a regular class. The class was chosen not to derive from monobehavior so that it can be serialized, which is important for data persistence.

Although it cannot be associated with a game object, it is possible to create an instance of *GameSettings* in another script, as we do with *GameControls*. For this purpose, a *GameManager* script was created associated with an object of the same name, which will be active from the moment the game starts until it ends.

In the *GameManager* script, an instance of *GameSettings* is created, making it accessible. The following code snippet shows the creation of the *GameSettings* instance in the *GameManager* script.

```
public class GameManager : MonoBehaviour
{
    public GameSettings gameSettings; //reference to gameSettings
```

```
        void Awake() //Called as soon as the object is created
        {
            gameSettings = new GameSettings(); //Creating the instance
        }
    }
```

Code Snippet 4.5: Creating the GameSettings instance

The *GameSettings* class is now public and accessible to all scripts from *GameManager*. To store the game settings, the following variables were created:

- **Music Volume** - integer value ranging from 0 to 100, and stores the current music volume value;
- **Sfx Volume** - integer value ranging from 0 to 100, and stores the current effects volume value;
- **Resolution** - integer value that stores the index of the resolution chosen by the player;
- **Fullscreen** - binary value that stores whether the player plays in fullscreen or not;
- **Quality** - integer value that goes from 0 to 2 and stores the quality index chosen by the player (0 - Low, 1 - Medium, 2 - High);
- **playerDevice** - binary value that stores whether player 1 plays with keyboard or gamepad.

Once an instance of the *GameSettings* class is created, the first step is to get the 16:9 aspect resolutions compatible with the player's monitor, these resolutions will be stored in a list, in order of increasing width. If the player monitor only supports 3 resolutions of 16:9 aspect, then the list will have 3 elements, which means that the *Resolution* index value can vary between 0 and 2.

The number of qualities that can be created is unlimited and their effects are manually defined by the developer in the Unity editor. In the case of *Overforged*, only 3 qualities were configured which are the qualities that come by default in the URP. Figure 4.13 shows the definitions of *High* quality.

Figure 4.13: High Quality Graphics Stettings

In the *GameSettings* class, functions for changing the settings were also added, more specifically:

- **ChangeMusicVolume(int direction)** - changes the music volume by 10 units, according to the direction. If the direction is -1, the volume decreases by 10 units, if it is 1, it increases by 10 units;
- **ChangeSFXVolume(int direction)** - changes the volume of the effects by 10 units, according to the direction;
- **ChangeResolution(int direction)** - changes to the previous or next resolution, according to the direction;

- **ChangeFullscreen(bool value)** - changes the window mode to Fullscreen, if the value is true, or to Windowed otherwise;
- **ChangeQuality(int direction)** - changes to the previous or next quality, according to the direction;
- **ChangePlayer1Device(bool value)** - changes player 1 device to gamepad if the value is true, or to keyboard otherwise.

In the function to change the resolution, after selecting the new resolution according to the direction, its width and height are sent as parameters to the *Screen.SetResolution(width, height, fullscreen)* function of Unity, this function will resize the screen and change the value of fullscreen. In this case, as the objective is only to change the resolution, the value of fullscreen sent is the value currently active.

For fullscreen, the process is similar, the function *Screen.SetResolution(width, height, fullscreen)* is used again, but this time, the width and height remain the same, sending only the new fullscreen value.

Changing the device of player 1 consists of changing only a binary value to true or false.

Changing volumes requires a more complex process. In unity, the sound is associated with an asset called *AudioMixer*. In the case of *Overforged*, we will only use one *AudioMixer* to control the sound of the entire game. This component has by default a single control group, called *Master*, this group controls the overall sound of the game. For the player to control the volume of music and sound effects individually, we create two subgroups of the *Master* group, *Music* and *SFX*. Figure 4.14 illustrates the *AudioMixer* of the game *Overforged*, where the subgroups created are observable.

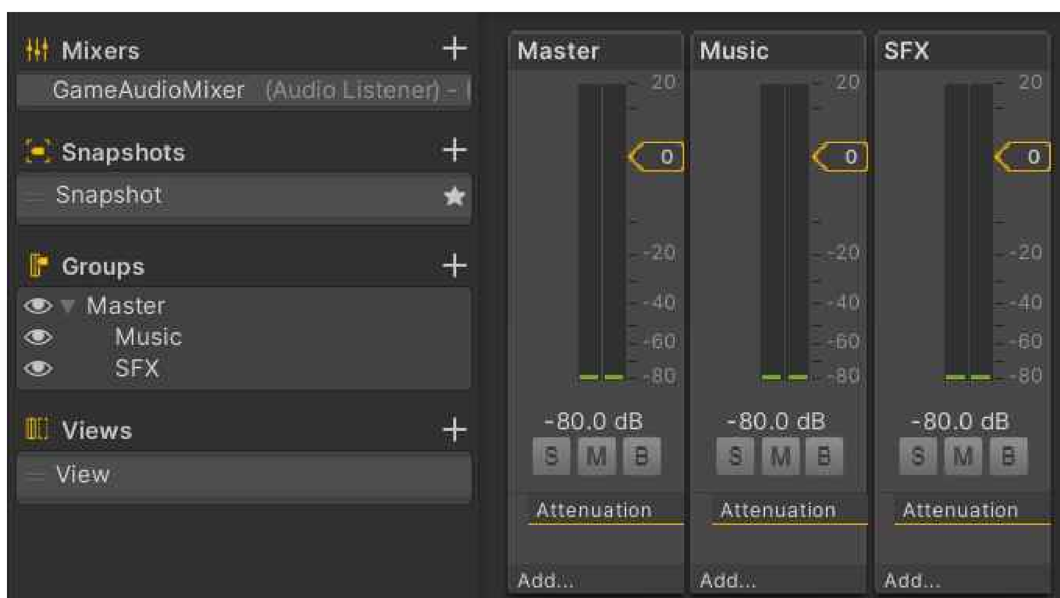

Figure 4.14: Overforged Audio Mixer

Each of the subgroups has an associated volume, but to be able to change that volume via script, we need to expose it as a parameter, right-clicking on the group and selecting

”Expose Volume Of Group To Script”.

As you can see in figure 4.14, the volume is a value that varies between -80dB and 20dB, but in our script, the volume varies between 0 and 100. To apply the values, we must subtract 80 to the volume value, so when the volume is 0, the value in *AudioMixer* is -80dB, and when the volume is 100, the value in *AudioMixer* is 20dB. The following code snippet shows the calculation performed to apply the music volume to *AudioMixer*.

```
int mixerVolume = (musicVolume - 80); //Because mixer goes from -80 to 20
audioMixer.SetFloat("MusicVolume", mixerVolume);
```

Code Snippet 4.6: Changing The Music Volume

4.2.5 Serialization

There are several ways to persistently store data in Unity like XML or SQL. In *Overforged*, the data will be saved through the Serialization method of the C# language.

Serialization is the process of converting an object into a stream of bytes so that it can be sent over the network or stored persistently. This method is more secure than the XML method because in XML the stored data can be easily manipulated.

To Serialize in a simple and modular way, the *SerializationManager* script was created, which has 3 static functions:

- **Save(string saveName, object dataToBeSaved)** - function that receives the name that the saved file must have, as well as its data;
- **Load(string saveName)** - function that receives the name of the file to be loaded;
- **Delete(string saveName)** - function that receives the name of the file to be deleted;

The functions are static so we can call them at any time without the need to create an instance of the class.

A *BinaryFormatter* is used for the writing and reading process, as a translator that has the necessary knowledge to transform objects into bytes and vice versa (Microsoft, 2022a). For the *BinaryFormatter* to know how to translate, it needs a Surrogate, which indicates exactly how to save and load a class (Microsoft, 2022c).

The code snippet below shows an example of a Surrogate for the *GameSettings* class, in which the *GetObjectData* and *SetObjectData* functions are responsible for saving and reading the data, respectively. In the function of saving the data, we instruct the Surrogate to save the class variables with a specific identifier, that will be used to retrieve the class variables as well.

```
        public void GetObjectData(object obj, SerializationInfo info,
            StreamingContext context)
        {
            GameSettings data = (GameSettings)obj;
```

```
        ...
        info.AddValue("resolution", data.resolution);
        info.AddValue("fullscreen", data.fullscreen);
        ...
    }

    public object SetObjectData(object obj, SerializationInfo info,
        StreamingContext context, ISurrogateSelector selector)
    {
        GameSettings data = (GameSettings)obj;
        ...
        data.resolution = (int)info.GetValue("resolution", typeof(int));
        data.fullscreen = (bool)info.GetValue("fullscreen", typeof(bool));
        ...
        obj = data;
        return obj;
    }
```

Code Snippet 4.7: Surrogate for the GameSettings class

When we save, load or delete data, we use the input/output functions of C# to check if the desired file exists, if it doesn't exist, we create a new one (in the case of saving and loading) or delete it (in the case of delete case). There must be a Surrogate for each class to be Serialized, and these Surrogates are used by *BinaryFormatter* to serialize the respective classes in the desired file. That file is stored in what Unity calls a persistent data path, that is, a location on each operating system that is immutable. For Windows, this location is the *AppData* folder.

In short, **when we want to save or load a class**, we first create a Surrogate and then freely **use the *Save* and *Load* functions of the *SerializationManager***.

4.2.6 Loading Screen

A loading screen is essential to any game so that the player has a less boring experience while waiting for the loading process to finish or at least to have an idea of the loading progress.

In *Overforged*, an example where a loading screen is used is when transitioning from the Main Menu to the World Map. While World Map elements and level data are being loaded, we want to show the players a loading screen with tips. The tips to be displayed on this screen are made of an image and a text, and remain on the screen for a certain time.

4.2.6.1 Loading Tips

The Tips will be created in data containers called *ScriptableObjects* (Unity, 2022d). For this purpose, we created a class named *LoadingTip* that derives from ScriptableObject instead of MonoBehavior, this class is made of the following three parameters:

- **tipText**: text shown to the player when the tip is on screen;

- **tipTime**: time in seconds that the tip remains on the screen;

- **tipIllustration** - picture shown to the player when the tip is on screen.

To create a data container derived from this class, we use the *CreateAssetMenu* attribute. The code snippet below shows the *LoadingTip* class:

```
[CreateAssetMenu(fileName = "Loading Tip", menuName = "Obj/LoadingTip")]
public class LoadingTip : ScriptableObject
{
    string tipText;
    float tipTime;
    Sprite tipIllustration;
}
```

Code Snippet 4.8: Loading Tip Class containing the different attributes of an loading tip.

With this, **we can now create several tips** that we can later use in a list of tips, to present them sequentially. Figure 4.15 illustrates an example of a created tip.

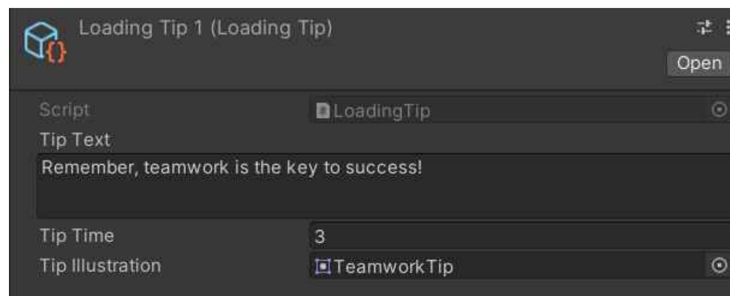

Figure 4.15: Example of an Loading Tip

4.2.7 The Loading Process

In Unity, to avoid strange behaviors or possible problems **there must always be at least a scene open**, to guarantee this, an empty scene named *PersistentScene* was created. This scene is the first scene in the game to be loaded and is never unloaded until the game closes.

To control the behavior of the loading screen, the script *LoadingManager* was created, in which we have the function of loading a new scene. In this function, we first present the graphical component of the loading screen, then unload the active scene asynchronously using the function *SceneManager.UnloadSceneAsync()* and load the new scene using the function *SceneManager.LoadSceneAsync()* using additive mode as loading mode. Additive loading mode means that the scene must be loaded without unloading the active scene.

The functions *LoadSceneAsync* and *UnloadSceneAsync* return an *AsyncOperation*, which we store in a list of operations, so we have a list of all load and unload operations to be performed. So, after we've instructed the unloading and loading of scenes, we start the execution of a coroutine. A coroutine is a function whose content is not guaranteed to run in a single frame, which means that performance impacts are minimal and that we will

not overload the main thread. In this coroutine, we wait for all operations to be complete, by checking if *operation.progress = 100*. When all operations are complete, we disable the visual component of the loading screen.

In *LoadingManager* we also have a list of tips, which is iterated in the coroutine. While we wait for the load operations to finish, we count the time since the last tip appeared, if this time exceeds the *tipTime* of that tip, we show the next one in the list. When reaching the end of the list of tips, we present the first one again. Figure 4.16 shows the behavior of the loading screen and the tips that are displayed on it.

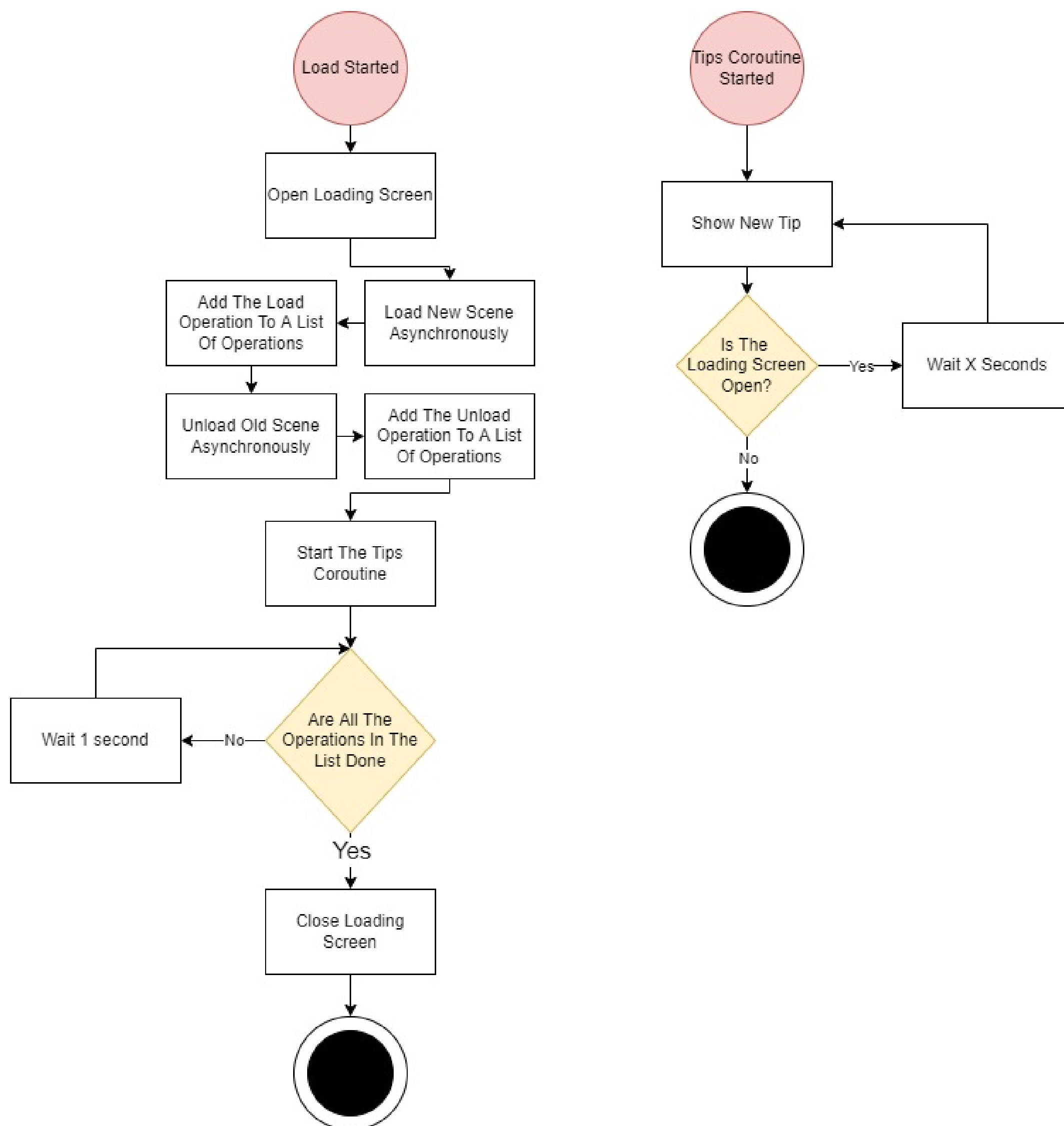

Figure 4.16: Loading Screen Behavior

Overforged's loading screen visual component consists of a canvas, with a background image, an image for the Tip's illustration, a text for the Tip's description, and a text saying "Loading...". Figure 4.17 illustrates the loading screen with an example of a tip.

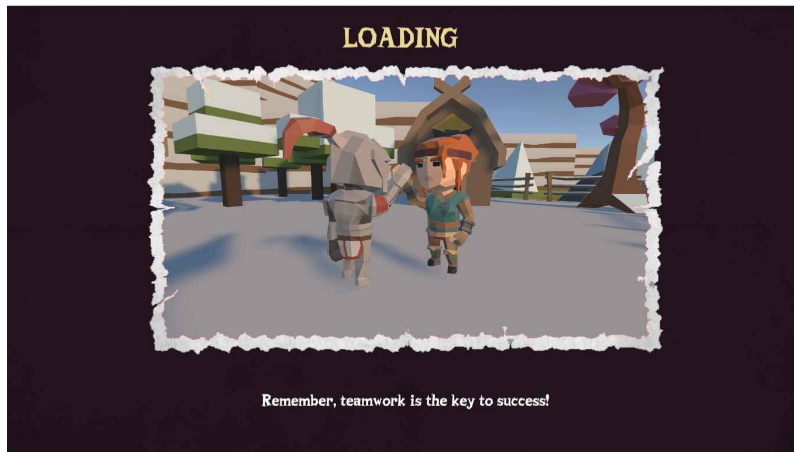

Figure 4.17: Loading Screen

Like this, **whenever we need to load a new scene, we just use the *Loading-Manager* functionalities**.

4.3 Main Menu

The *Main Menu* of the game consists of a set of interconnected menus through which the player can start the game, change his character or configure the game's graphics and sound settings.

As mentioned in section 4.2.2, the *Main Menu* is a unique type of level, so to control that level we created the *MainMenuLevelManager* derived from the *LevelManager* earlier prepared, so we have access to the players' references and the exchange functions between them. We also added to *MainMenuLevelManager* a reference to the first screen of the *Main Menu*, the *PromptScreen*, which should be opened as soon as the scene is opened, by calling the *Open()* function on it.

The *Main Menu* will be structured as shown in figure 4.18.

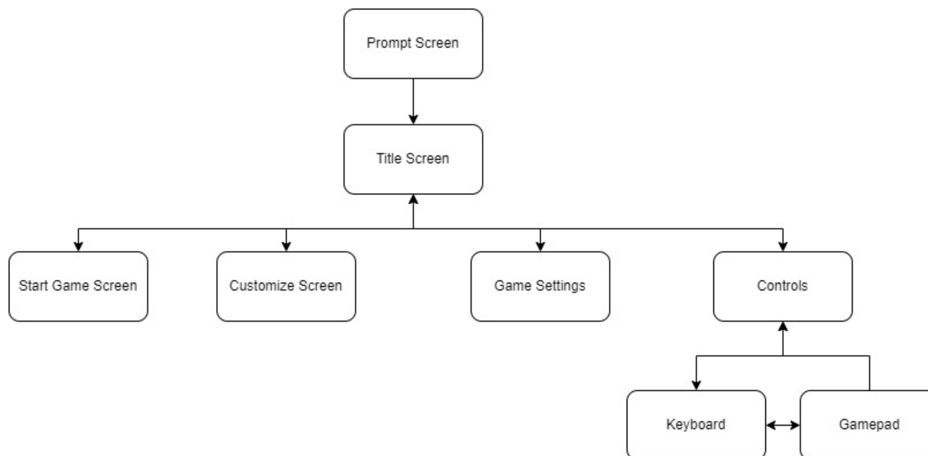

Figure 4.18: Main Menu Structure

Each of the menus shown in figure 4.18 was built on an individualized Canvas, whose rendering mode is *World Space*. In all menus, both players are visible, as well as the

entrance to a forge. These menus are essentially composed of simple buttons, except the *GameSettings* menu which needs more complex buttons. All menus are developed almost identically, so we will only analyze the *TitleScreen* menu.

The *TitleScreen* has four buttons, the *StartGame* button, the *Customize* button, the *Settings* button, and the *Controls* Button. Each of these buttons takes the player to a different menu. Figure 4.19 illustrates the *TitleScreen* menu of the Main Menu, where the players and the workshop entrance are visible.

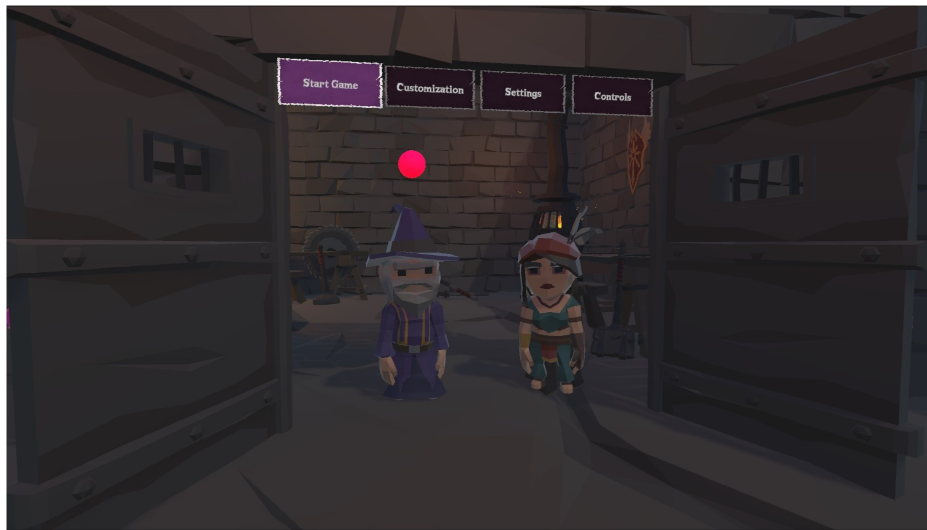

Figure 4.19: TitleScreen menu from the Main Menu

This menu, like all the others, derives from the previously developed script *UI_Menu*, inheriting the navigation, confirm and back functions. In addition to the inherited functions and parameters, references to the buttons *Start Game*, *Customization*, *Settings*, and *Controls* are added to the script of this menu.

When we open the menu using the function *Open()*, we subscribe to the events *OnButtonHovered* and *OnButtonClicked* of the 4 buttons on the menu, saying which functions to call when these events are fired. For instance, when the mouse hovers over any of the three buttons, we call a function that puts the previously hovered button in the normal state and puts the new button in the hovered state. When we click the *StartGame* button, we call a function that closes the current menu and opens the *StartGame Screen* menu. The code snippet below shows the subscription to the events of the 3 buttons and the functions associated with them.

```
//Subscribe to the hover event and the clicked event to each button
startGameButton.OnButtonHovered = () => OnButtonHovered(startGameButton);
startGameButton.OnButtonClicked = () => OnStartGameClicked();

customizeButton.OnButtonHovered = () => OnButtonHovered(customizeButton);
customizeButton.OnButtonClicked = () => OnCustomizeClicked();

settingsButton.OnButtonHovered = () => OnButtonHovered(settingsButton);
settingsButton.OnButtonClicked = () => OnSettingsClicked();
```

```
controlsButton.OnButtonHovered = () => OnButtonHovered(controlsButton);
controlsButton.OnButtonClicked = () => OnControlsClicked();
```

Code Snippet 4.9: Subscribing to button events

The configuration of the neighborhoods of the four buttons, so the player can use the keyboard to navigate between them, will be as follows:

- **Start Game** - has the *Customization* button as a neighbor when navigating to the right;
- **Customization** - neighbors the button *Start Game* when navigating to the left and the button *Settings* when navigating to the right;
- **Settings** - neighbors the button *Customization* when navigating to the left and the button *Controls* when navigating to the right;
- **Controls** - neighbors the button *Settings* when navigating to the left.

Figure 4.20 illustrates the list of neighbors for the *Customization* button in the Unity editor.

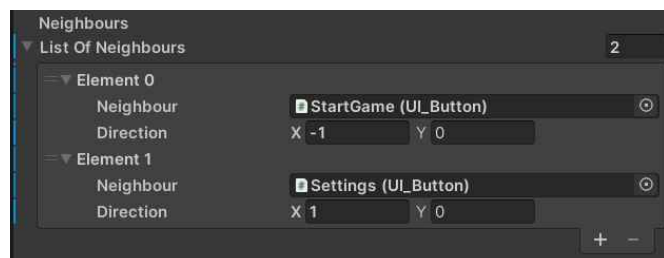

Figure 4.20: Neighbours of the Customization button

4.3.1 Settings Screen

The *Settings Screen* menu is made up of unique buttons, made up of other buttons. These buttons, in addition to performing functions, also show some information. Figure 4.21 illustrates the *Settings Screen* menu, where we can see 3 different types of buttons: the button with bars, the button with text options, and the button with a checkbox.

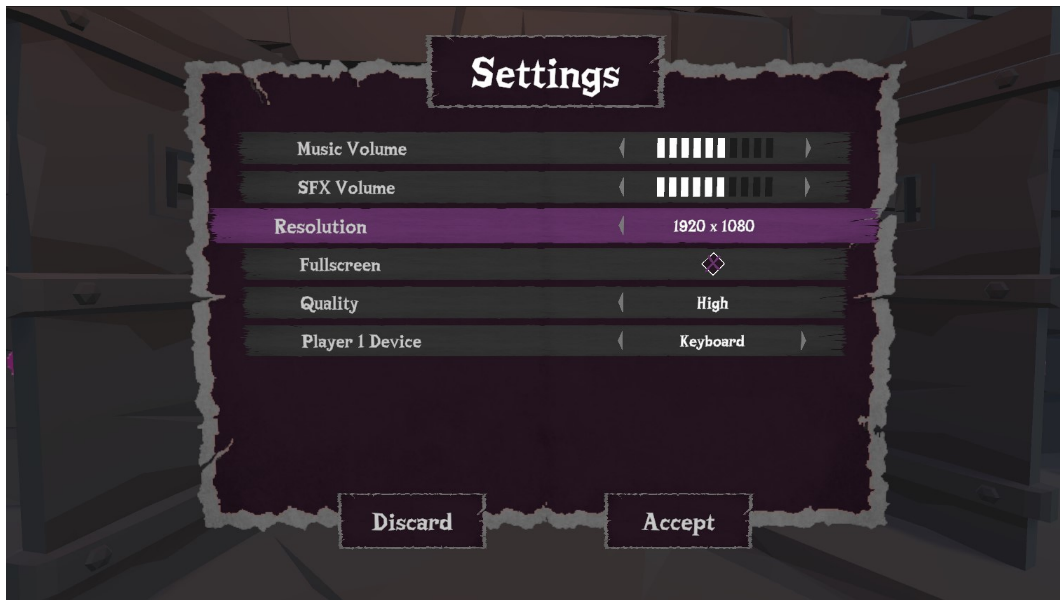

Figure 4.21: Main Menu Settings Screen

As shown in the figure, all these new types of buttons are made of other buttons inside them. For example, the button *Music Volume* is made of two arrow buttons, which serve to decrease and increase the volume.

The *Settings Screen* menu, like all other menus, has an associated script, which derives from the base *UI_Menu* script, but in this script, as soon as the menu is opened we keep a copy of the current settings in case the player chooses to discard the changes.

When subscribing to button events, we have to subscribe to many more events than in other menus, taking the *Music Volume* button as an example, for this one, we will have to subscribe to the *OnButtonHovered* event of the big background button, but not on the its child buttons that represent the two arrows, as we are only interested in the visual feedback on the big background button. In the *OnButtonClicked* event, we are only interested in subscribing to this event in its child buttons that represent the two arrows, since the click on the big background button is irrelevant. The code snippet below shows the subscription of events for the *Music Volume* button.

```
musicVolume.onHoveredDelegate = () => OnButtonHovered(musicVolume);
musicVolume.leftArrow.onClickedDelegate = () => ChangeMusicVolume(-1);
musicVolume.rightArrow.onClickedDelegate = () => ChangeMusicVolume(1);
```

Code Snippet 4.10: Subscribing to ”Music Volume” button events

In this button, we also have 10 black bars and 10 white bars, that will be active or inactive depending on the value to be displayed. If the current volume is 80, then we will activate the first 8 white bars and the last 2 black bars.

The other buttons work in the same way, using the functions implemented in the *GameSettings* script, described in the 4.2.4 section, varying in small aspects and in the type of presentation they show to the player.

At the bottom of the screen, there are 2 buttons, which are assigned the function of accepting the new settings or discarding them and returning to the previous settings. By

discarding the changes, we change the settings so that they have the values of the copy of the settings created when opening the menu. When accepting the definitions, we use the *Save* function of the *SerializationManager* described in section 4.2.5 to save the new definitions.

4.3.2 Customization Screen

Another different menu is the customization menu. This menu has two buttons on top, the *Accept* button and the *Discard* button, to accept and reject the changes. When the customization screen is opened, the index of the model currently in use is stored, so if the player chooses the Discard button, we have a copy of the index from the player's old model. There are two arrow buttons on each player that allows changing its model. Figure 4.22 illustrates the customization submenu for player 1.

Figure 4.22: Customization Menu

As described the section 4.2.1, the player has an associated script named *Player-Model*, in this script, there is a reference to the player models in a list format. When the player changes the model in one direction, we simply deactivate the current model and activate the next/previous one.

4.3.3 Controls Screen

The last one is the controls menu. This menu has a panel with an image of the controls in it, and two buttons on the bottom, one to change between the keyboard and the gamepad controls, and the other to go back to the main menu. When the controls screen is opened, the image on the panel shows the gamepad controls and the change button has the text ”Keyboard” on it. When clicking that button, the gamepad controls menu is close, and the keyboard controls menu is opened. These two menus are a copy of each other,

differing only in the image of the controls they show, and the text of the change button. Figure 4.23 illustrates the controls menu for the gamepad.

Figure 4.23: Controls Menu

4.4 World Map

The Main Menu is completed and the next step was the development of the World Map. **The World Map is the scene where the player will choose the level he wants to play** and where he will be able to consult the scores he obtained in the levels.

4.4.1 World Map Environment

To build the World Map Environment the package ”POLYGON MINI - Fantasy Pack” from Synty Studios was acquired and imported. This pack can be found on the Synty Store (Synty, 2022b) and it offers a vast set of environment assets of different themes. Figure 4.24 represents the ”POLYGON MINI - Fantasy Pack” from Synty Studios, used for the World Map Environment.

Figure 4.24: POLYGON MINI - Fantasy Pack, Synty Studios, from Synty Store (Synty, 2022b)

Using the resources available in the package, the World Map essentially consists of several biomes, representing the journey that the player takes through different parts of the world to complete his task. Pins were placed throughout the map, which represents a level. Figure 4.25 illustrates the World Map of the game Overforged.

Figure 4.25: Overforged world map

4.4.2 The Level Logic

On the World Map, **navigation between levels must be automatic according to the chosen direction**, that is, if the player is at level X and tries to move to the right, locomotion from level X to level Y must be automated by the game.

The first step to implementing the desired features was the creation of the *LevelPin* script, which will contain the information and features of the levels.

Within this script, a neighborhood system identical to the one created for the buttons explained in section 4.2.3 was implemented. The system consists of a reference to the neighbor level and a Vector2 with the direction to that neighbor. Also, the following variables and functions were added to the *LevelPin* script:

- **playerStandingPoint** - point where the playable character should stand when the player moves to the level;
- **cameraStandingPoint** - point where the camera should be positioned when the player moves to the level;
- **listOfNeighbours** - list of neighbors of the level;
- **FindNeighbourOnDirection(Vector2 direction)** - returns the neighbor level of the list of neighbors according to the direction sent as a parameter. Returns null if there is no neighbor in that direction.

With this, each level has a list of other levels it connects to as well as the direction to it.

To move between levels we use a Pathfinding component included in Unity, the *NavMeshAgent* component. **The *NavMeshAgent* can be used to find the shortest path between two points** according to the surfaces on which it can move. In *Overforged's* World Map **we only want the player to move in predetermined paths**, so the simplest way is to make only these paths usable by *NavMeshAgent* (Unity, 2022c).

A mesh is essentially usable by *NavMeshAgent* if its object is marked as *NavigationStatic*, so to ensure that the shortest path found is also the only existing path, we will mark as *NavigationStatic* a predefined set of objects. Figure 4.26 illustrates how to make a Navigation Static object.

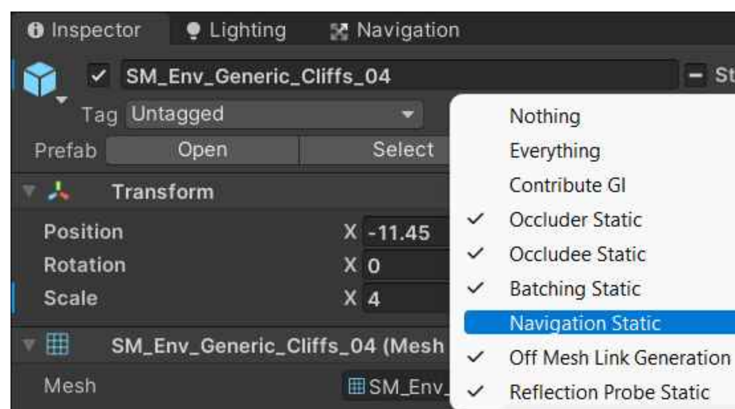

Figure 4.26: Marking an Object as Navigation Static

The object that the player can walk under consists of an invisible flattened cube that will be marked as Navigation Static and a white stretched cube that will be used to form a dashed line. This object, called *Navigation Unit* is illustrated in figure 4.27.

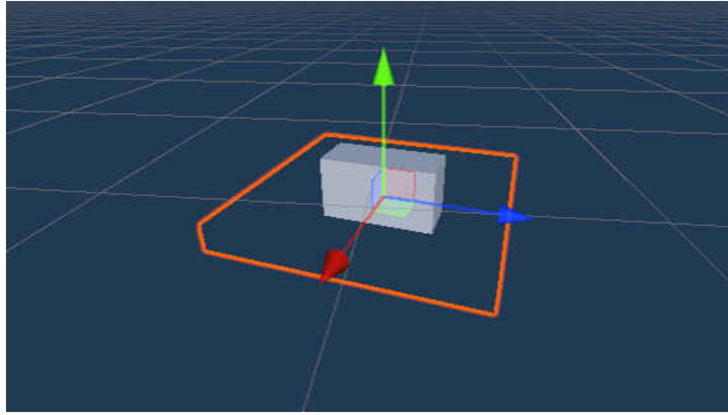

Figure 4.27: World Map Walkable Unit

By chaining together a group of Navigation Units we obtain a path whose visual representation is a dashed line. Figure 4.28 illustrates a path between two levels formed by a set of Walkable Units.

Figure 4.28: World Map Path Made Of Walkable Units

For this system to work, we need a script to store a reference to the pin the player is on and to detect inputs. This script is named *WorldMapLevelManager* and derives from the *LevelManager* described in 4.2.2, inheriting all its functionalities but we also add a reference to the current pin. In each frame, we check the direction in which the player wants to move and according to the direction we move the player to a new pin.

4.4.3 World Map Player

The World Map player is the most different from the other types. The script *WorldMapPlayerManager* was created for this type of player, which derives from the script *PlayerManager*, inheriting all the attributes and functions described in section 4.2.1. In addition to the base components, a second *PlayerModel* has been added to this player that repre-

sents the Player 2 character, as well as references to its animator to play animations in both models.

The *NavMeshAgent* component was also added, which will take advantage of the pathfinding algorithm offered by Unity.

Finally, to control the player's movement, the function *SetLocomotionTarget(Vector3 newTarget)* was added to the *WorldMapPlayerManager* script that changes the target of *NavMeshAgent*.

Finally, a model of a boat included in the package "POLYGON MINI - Fantasy Pack" was added as a child object of the player, which will be useful later to navigate in the water zone. Figure 4.29 represents the World Map Player, with the two models.

Figure 4.29: World Map Player

4.4.4 World Map Interactables

Figure 4.25 shows us that the map is made of zones of different heights and also water zones. These zones are useful to diversify the content of the World Map and the movement of the player, adding animations and objects that allow reaching these zones, some of those objects are:

- **Ladder** - ladder used to go up one zone;
- **Boat** - used for the player to move in the water.

When moving between two levels and the target zone is above the current one, the *NavMeshAgent* will move as close as possible, but **it won't be able to reach**

its destination until the player is placed in the top zone. **To solve this problem the Ladder was implemented**.

A ladder has two trigger colliders, one at the top and one at the bottom, these colliders are responsible for detecting the presence of the player and triggering an action. For instance, when the player touches the collider at the bottom of the ladder, we start the climb it up, and then when he touches the collider at the top of the ladder, we finish the climb. The diagram represented in figure 4.30 represents the logic used to move the player along the ladder using the two colliders.

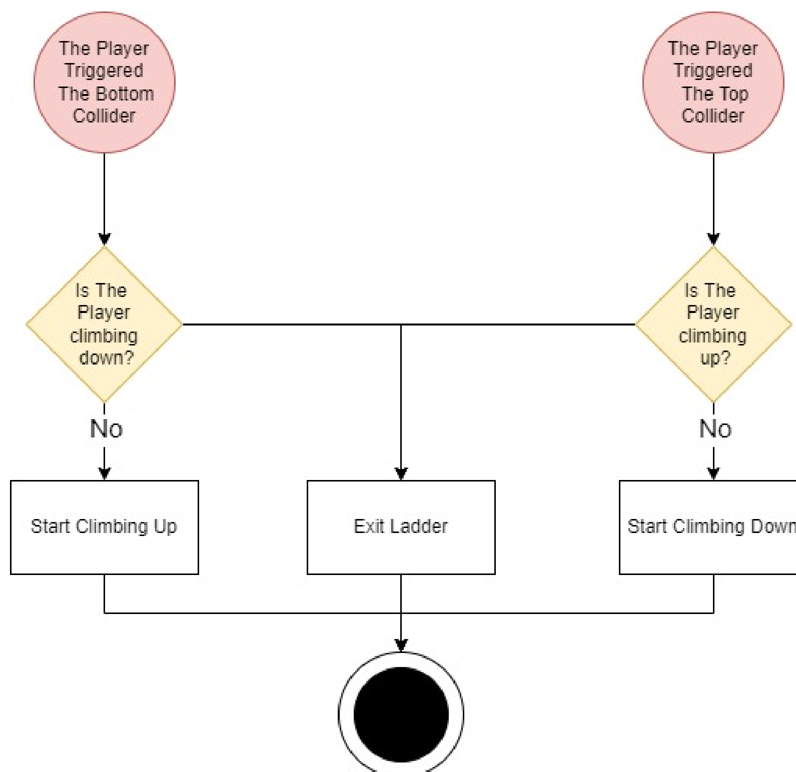

Figure 4.30: Ladder Logic

As the player climbs the ladder, new animations are played on the two models, in which the first player climbs the ladder and the second player climbs onto the first player's back. Figure 4.31 illustrates the animations played by players when climbing the ladder.

Figure 4.31: Player animations played when climbing the ladder

The boat works similarly, to switch between walking and riding the boat, several colliders were placed on the map, when the player collides with one of these colliders, there are two possibilities:

- the player is on foot mode and therefore will switch to boat mode;

- the player is in boat mode and therefore will switch to on-foot mode;

Switching to boat mode consists of activating the boat model and playing two animations for the two models. Switching to on-foot mode consists of deactivating the boat and playing the locomotion Blend Tree. Figure 4.32 illustrates the World Map Player's boat mode.

Figure 4.32: Boat Mode of the World Map Player

4.4.5 World Map UI

To show players the score they obtained in the level they are on, **we used a panel that can appear in one of the 4 corners of the screen**, depending on the level. In this panel, the player can observe the name of the level, an illustration of it, the score, and the number of obtained stars.

If the score is greater than or equal to the required score for a given star, it will be drawn in a golden color, otherwise, it will be drawn in dark gray color.

Whenever a player moves to a certain level, the panel is repositioned to a predefined corner of the screen, and new information is placed on the panel. The panel consists of a Canvas with the necessary image and text elements, as shown in figure 4.33.

Figure 4.33: UI Panel With a Level Information

When the player presses the pause button, a panel with three buttons is displayed on the screen, and any movement and action inputs are ignored and only menu navigation inputs are valid. In the pause menu, there are three options, resume, settings, and exit to the main menu. The resume option simply closes the pause menu, the settings button opens a game settings panel and the exit option loads the Main Menu. Figure 4.34 illustrates the WorldMap level pause menu.

Figure 4.34: World Map Pause Menu

Figure 4.35 illustrates the Pause Menu Settings.

Figure 4.35: World Map Pause Settings

4.5 Playable Levels

The levels placed on the world map are the playable levels, in which the player must be able to deliver as many orders as possible in a predefined time, to obtain the best score he can. The playable levels are complex and rich in interactions and obstacles.

As a first step in the development of playable levels, it is necessary to inherit the basic level variables and functions mentioned in the 4.2.2 section, therefore, the *SmithingLevel-Manager* script was created, which derives from the script *LevelManager*. In this script, the time, score, and order management systems will also be configured.

4.5.1 Tutorials

Each playable level requires the forging of a different weapon that a new **player does not know how to forge**. It is therefore important to **offer the player a brief tutorial**, before starting the level, that shows the steps that he must follow.

The tutorials consist of a text corresponding to the title of the tutorial, another text corresponding to the description of the tutorial, and an image with the illustration, as shown in the following code snippet.

```
public class Tutorial
{
        string tutorialTitle;
        Sprite tutorialIllustration;
        string tutorialDescription;
}
```

Code Snippet 4.11: Class "Tutorial" containing the tutorial information.

In the *SmithingLevelManager* script, a list of Tutorials was added, containing the various tutorials of that level. The tutorials will be presented in a menu, with a panel containing the information of the tutorial and two buttons to move to the next or previous tutorial and one button to close the panel.

When in the first tutorial, only the forward button is displayed. When advancing, the information of the following list element is displayed and the displayed buttons are updated. If the new tutorial is the last one, the "Close" button is shown in the same position as the "Next" button.

Once the level is loaded, the tutorial is shown, and only after closing the tutorial does the level start. Figure 4.36 represents the tutorial screen.

Figure 4.36: Tutorial Screen

4.5.2 Time, Score And Oders

Each level has a time and a set of scores for the 3 stars. To act as a time counter, a real number was defined which is decremented every second. This decrementation occurs in a *Coroutine* which is a function that can be paused and resumed. In this *Coroutine*, the counter is decremented by 1, the execution is paused for 1 second, and the process is repeated until the counter reaches zero. Figure 4.37 represents the operation of the *Coroutine* used to decrement the time.

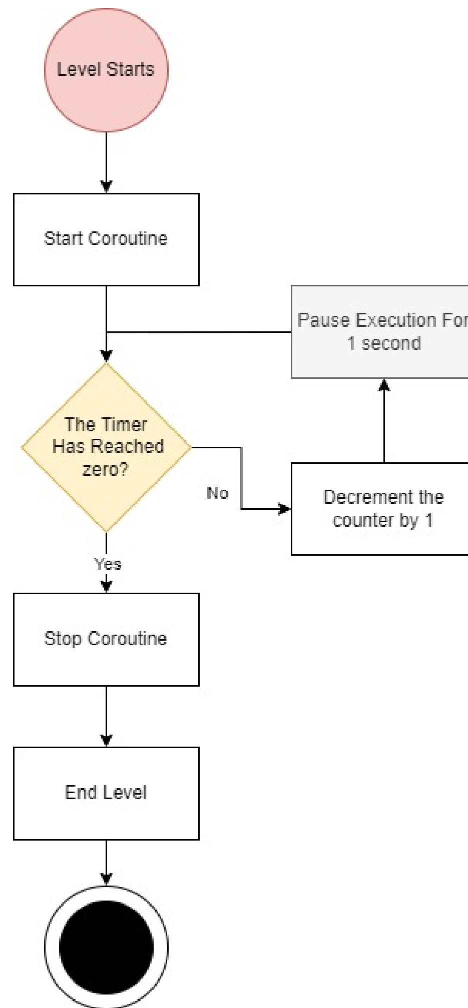

Figure 4.37: Level Timer Coroutine Logic

The main goal of the game is to prepare the requested orders in the given time. Each order has a set of information that defines it and to store this information, the *Order* class was created. This class is responsible for storing the necessary information, such as the object to be delivered, the time limit, the remaining time, and the score. The code snippet below represents the *Order* class.

```
public class Order
{
    public Interactable orderInteractable;
    public int orderScore;
    public float timeOnScreen;
    public float remainingTime;
}
```

Code Snippet 4.12: Class ”Order” containing the different attributes of an order.

Each order will have a score, more complex orders will be worth more points, and orders that are not on the screen (if the player decides to forge something random that is not one of the orders in the level) are not worth points. **Each valid delivered order will be worth a certain set of points, but likewise, if the player fails to deliver**

that order in time, he will lose that same set of points.

To handle the delivery and failure of orders, the functions *DeliverOrder(Order order)* and *FailOrder(Order order)* were created, which receive the details of the order as a parameter.

In the *DeliverOder* function, the player's score is incremented and the order is removed, as it is already completed, while in the *FailOrder* function, the player's score is decremented, and the order is also removed because it has expired. The player's score is stored as an integer number.

Each level has a list of possible Orders, called *AllOrders*, as well as a list of Orders to be displayed on the screen, called *ScreenOrders*. The ScreenOrders list is empty at the beginning of the level.

Every second, inside the Coroutine responsible for decrementing the timer, we check if the number of Orders on the screen is less than 4, and if it is, the first Order is removed from the AllOrders list and added to the ScreenOrders list. Also within Coroutine, every second the remaining time of the 4 orders on the screen is decremented and when it reaches zero, the FailOrder function is executed, removing the order from the ScreenOrders list and placing it back on the AllOrders list. If an order is successfully delivered, the DeliverOrder function is executed, and the same list swap happens. This process is repeated until the level timer reaches 0. Figure 4.38 illustrates the order management process.

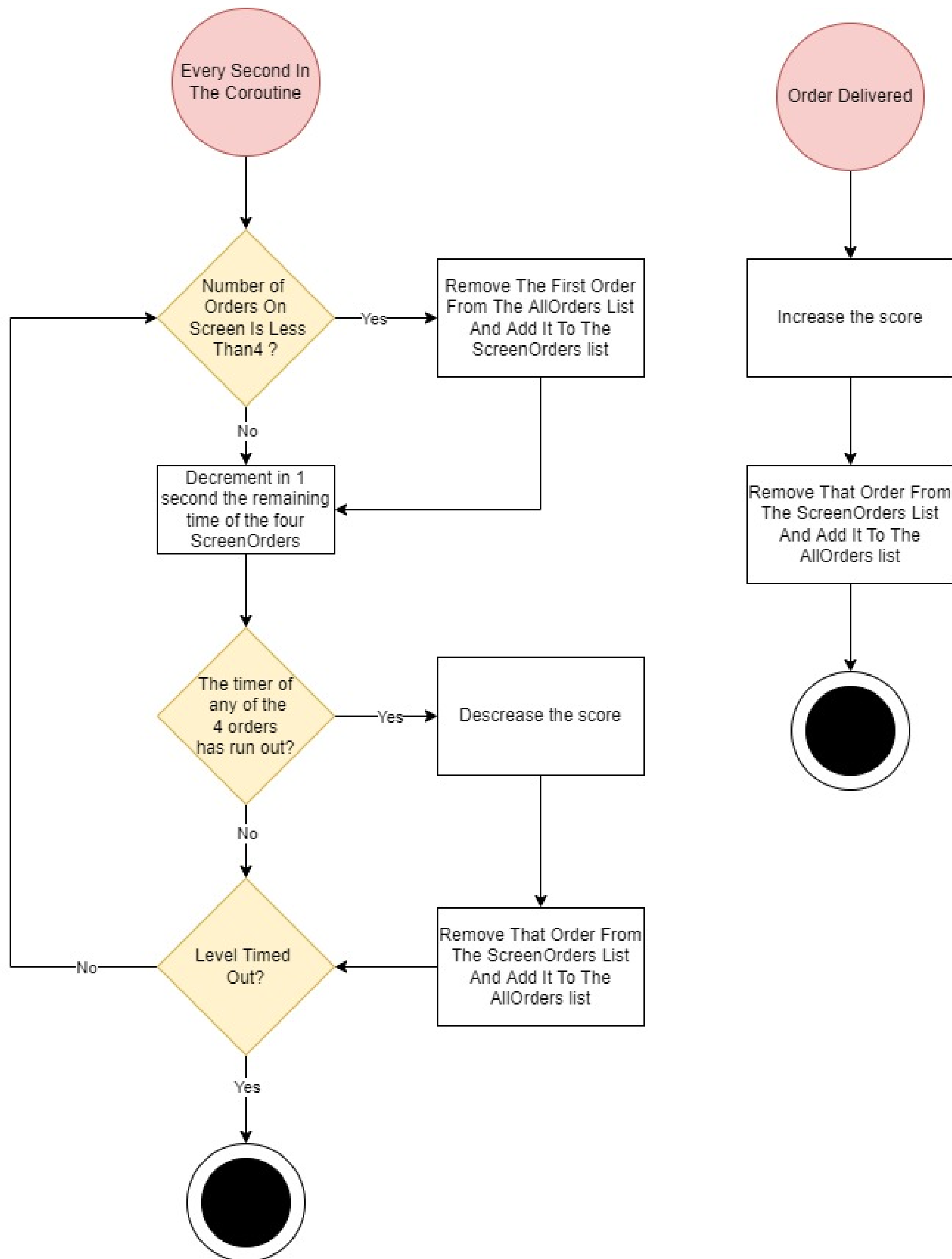

Figure 4.38: Orders Logic In The Playable Levels

4.5.3 The Playable Level UI

To provide the player with visual feedback on Orders, time, score, and other necessary elements, we created a set of essential elements in the HUD, a pause menu, and a end level menu.

The HUD elements consist of a panel with the remaining time of the level, a panel with the score obtained, and a set of 4 panels representing each of the 4 orders of the level. Every second, in the *Coroutine* responsible for decrementing the level timer, the content of these elements are updated, that is, the time and score text are updated to correspond to the level's time and score, and the data of the order panels is updated to match the four

orders in the ScreenOrders list.

Each panel that represents an order is made of an illustration of the deliverable, as well as the remaining time, represented in the *UI Slider* format referred to in the 4.2.3.1 section, and the ingredients to make that deliverable. Figure 4.39 illustrates the HUD elements of playable levels.

Figure 4.39: Playable Levels HUD

Like WorldMap, playable levels also have a pause menu. When this menu is opened, the *Coroutine* is paused until the menu is closed, thus preventing the timers from decrementing and ignoring any input not related to UI navigation. The resume option resumes the *Coroutine* and closes the menu, the settings options opens the Settings Panel and the Exit option takes the player to the world map, discarding any scores obtained. The Settings Panel is the same as the one mentioned in section 4.4.5. Figure 4.40 illustrates the Pause Menu.

Figure 4.40: Playable Levels Pause Menu

At the end of the level, when time runs out, a panel is presented to the player, with the player's score, the number of stars obtained, as well as the details of orders delivered and failed. In addition to the information panel, two buttons are placed on the screen, one that allows the player to continue to the WorldMap and the other one to replay the level. Regardless of which player selects, the score obtained is saved. Figure 4.41 illustrates the score panel of the playable levels.

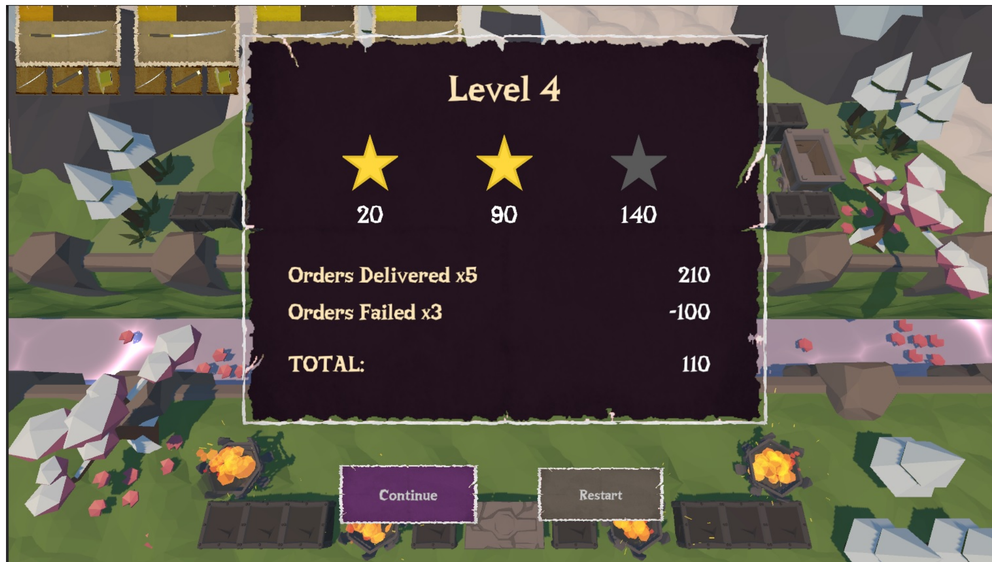

Figure 4.41: Playable Levels Score Panel

4.5.4 Playable Levels Persistent Data

Using the method described in the 4.2.5 section, the *GameProgress* class was created, which will be used to store data related to playable levels. Inside the class we have:

- A list of integer numbers, where each element of the list corresponds to a level, and the value of the element corresponds to the highest score obtained in the level;
- An integer containing the index of the last played level;

Whenever we complete a level, if the score obtained is greater than the previous high score, then we store the new score in the element of the score list corresponding to the level.

When we open the world map, we place the players on the pin corresponding to the last level played, so that the player does not have to redo the entire route. Even when opening the world map, depending on the high scores saved, we unlock or block the paths between the levels and calculate the stars for each level.

4.5.5 Playable Levels Player

The *PlayableLevels Player* is the most complex among the developed players. The script *PlayableLevelPlayerManager* was created for this player, which derives from the

script *PlayerManager*, inheriting all the attributes and functions described in section 4.2.1.

The first mechanic to develop is movement. **The movement must be relative to the level's camera** so that it is intuitive for the player. Player movement inputs are made of two axes, the y-axis and the x-axis corresponding to the vertical and horizontal movement input, respectively. These inputs can take any value between -1 and 1, but never outside these limits. To ensure that the player's movement is relative to the camera, we multiply the x-axis by the side vector of the camera, and the y-axis by the frontal vector and finally, we add the two vectors, obtaining the intended movement vector.

For instance, if the player presses the W and A keys simultaneously, this results in a movement input with the value 1 on the Y-axis and -1 on the X-axis. Now, we multiply the X-axis value by the side vector of the camera (the side vector of the camera represents its right), therefore, as the value of the input X-axis is -1, the result is a vector that points in the opposite direction, that is, its left. Next, we multiply the Y-axis value by the forward camera vector (the forward vector represents its front), and therefore, as the Y-axis value is 1, the vector does not change and the result is the camera's forward vector. Adding the two resulting vectors together, we get a diagonal vector that points to the forward left of the camera. Figure 4.42 represents the given example, where we can see a camera and a player (red cube), and three arrows. The black arrows are the result of multiplying the X-axis by the side vector and the Y-axis by the forward vector, and the white arrow represents the sum of the results.

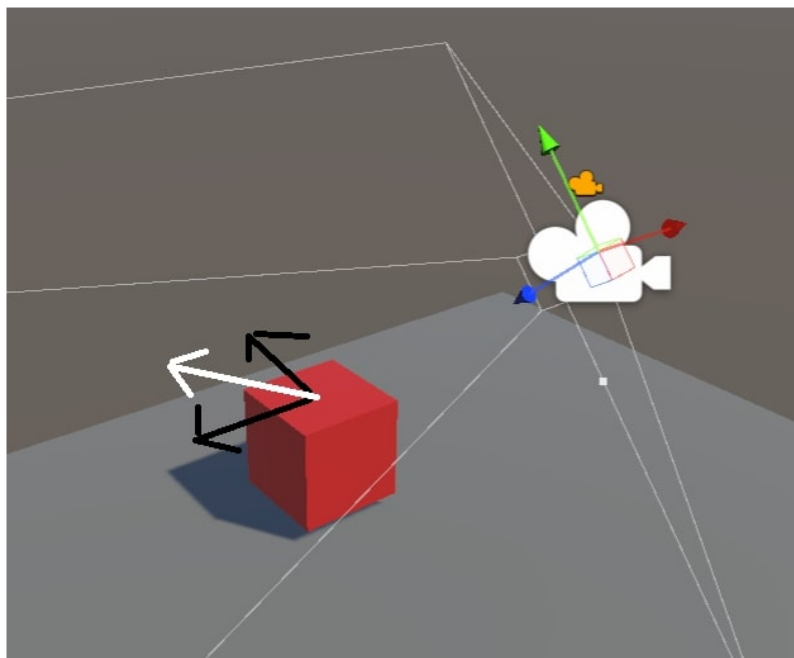

Figure 4.42: Calculation Of The Movement Vector Based On The Camera

The result of this calculation has a problem. The calculated movement vector is of type (x,y,z), where ideally the value of y would be zero, as we don't want the player to go up or down. In situations like the one illustrated in figure 4.42, in which the front vector of the camera points slightly downwards, the player will also move slightly downwards.

To prevent this unwanted movement, we remove any value in the y component of the resulting vector, thus obtaining a movement vector of type (x,0,z).

After calculating the motion vector, it is necessary to check if the player is touching the ground, which is done through a method called *Raycast* that consists of casting a ray from a point of origin, with a specific length and in a specific direction (Unity, 2022e). In this case, a ray is cast from the player's feet, with 0.5 units in length, directed downwards. If this ray hits a collider, it means the player is on top of something and therefore is grounded, otherwise, the player is in the air. Figure 4.43 illustrates the two possible results of the raycasting method. On the left, we have an example where the casted ray hits a surface and therefore the player is grounded, in the example on the right, the ray does not hit any surface and therefore the player is in the air.

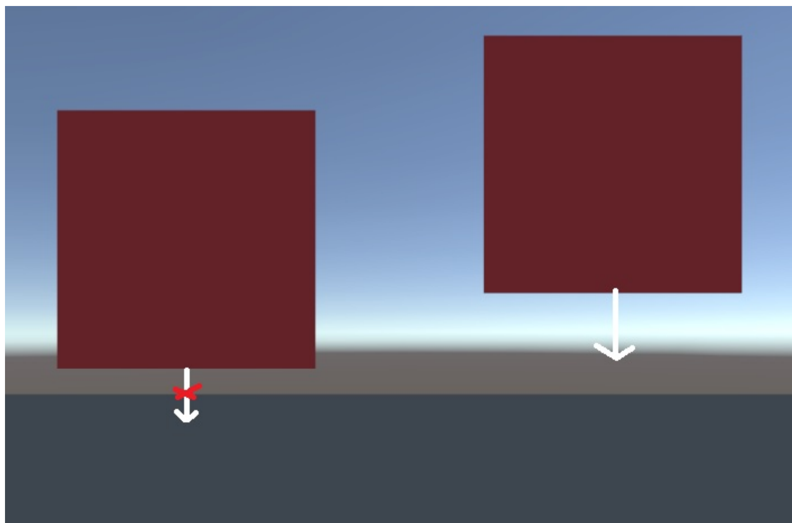

Figure 4.43: Ground Check Using Raycast

It is the ground check that will decide the value of the y component of the motion vector. If the player is on the ground, the Y component will keep the value 0, if the player is in the air, the Y component will decrease by a specific gravity force each Update cycle. For example, if the gravity force is 0.5, and the player stays in the air for 10 frames, then at the end of those 10 frames, the y component of the motion vector will have the value -5.

With the motion vector calculated, we can apply it to the player's Rigidbody, changing its velocity component, consequently moving the player. The following code snippet represents the assignment of the motion vector to the speed of the Rigidbody.

```
rigidbody.velocity = movementVector;
```

Code Snippet 4.13: Changing the Rigidbody Velocity

In addition to moving, the player can perform the action ***Dash***, which consists of **quickly moving the player in a certain direction**. The Dash can be implemented in several ways, such as associating an immediate force to the Rigidbody, using the *AddForce* method. This method is more physically realistic but opens the door to more unwanted and unexpected cases due to the physics system. Another method is using an animation with *Root Motion*, which is an animation that translates and rotates the player object.

Normally, when an animation is played, the player object remains at the same point and with the same rotation, and only the model moves. To move the entire player object along with the animation. We activate the *Use Root Motion* mode of the *Animator* component.

When the player presses the input of the Dash action, the animator's *Use Root Motion* mode is activated, and at the end of the animation being played, the mode is deactivated again.

During the Dash, we want to emit some particles to simulate dust and to reinforce the dash action. For these particles and any other particles present in the game, the package *POLYGON - Particle FX Pack* from Synty Studios was used. This pack can be found on Synty Store (Synty, 2022c) and is represented in figure 4.44.

Figure 4.44: POLYGON - Particle FX Pack from Synty Studios, from Synty Store (Synty, 2022c)

The dash particle is played for 1.5 seconds and then is automatically destroyed. Figure 4.45 illustrates the Dash action, along with the animation and associated particles.

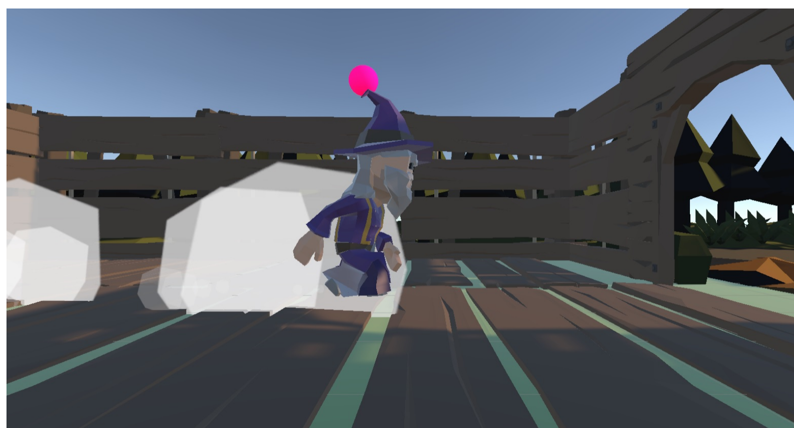

Figure 4.45: Dash Action

4.6 Interactables

In *Overforged*, the interactables in the game are complex and very different, but even so, they share a large set of features and variables, among which:

- **Rigidbody** - component to simulate the physics of a rigid body;
- **BoxCollider** - component to make the interactable collide with other colliders;
- **interactableName** - name of the interactable;
- **interactableIcon** - representative illustration of the interactable;
- **audioSource** - audio source of the interactable;
- **litColor** - color of the interactable material emission in the lit state;
- **interactableMeshes** - list of meshes that make up the interactable;
- **LitMe()** - function that changes the color of the meshes that make up the interactable to litColor;
- **UnlitMe()** - function that changes the color of the meshes that make up the interactable to its original color;
- **HasConditionsToInteract()** - function that returns a true flag if the player can interact with the interactable or a false flag otherwise.

The functions *LitMe()* and *UnlitMe()* change the *Emission* component of the materials that make up the meshes of the interactable. The emission component is generally used to make the material look like a light source that can be activated and deactivated, and is ,therefore, the ideal component for this type of use. The emission consists of a color accompanied by an intensity level, the greater the intensity, the greater the light effect that the material emits. If the color intensity is a low value, then the result obtained is the original material with a little bit of emitted light. The function *LitMe()* activates the emission component of all the materials of all the meshes of the interactable and assigns it the color of the variable *litColor*, while the function *UnlitMe()* disables the *emission* component.

Figure 4.46 illustrates the example of an interactable in its Lit and Unlit state. On the left is the original state of the interactable, that is, when the material does not have the emission enabled and on the left, we have the result of the *LitMe()* function, where we can see that the material has the emission enabled and it looks like emit some light.

Figure 4.46: Interactable Lit and Unlit

For the player to detect interactables, the script *PlayerInteractions* was added to the player, in this script, using the *OnCollisionEnter* function built into Unity, we detect which interactable is in front of the player. Upon detecting that interactable, the function *HasConditionsToInteract()* is called and if it returns the *true* flag, a reference to the interactable is stored in the script.

The interactables are divided into two main types, the *MovableInteractable*, which is an interactable that can be moved, either by the player or by physics, and the *StaticInteractable* which cannot be moved. These two interactables derive from the base interactable, inheriting all the variables and functions mentioned above.

4.6.1 Movable Interactables

A movable interactable, in addition to the base interactable, has the following variables and functions:

- **canBeThrown** - flag that assumes the value true or false, which means that the object can or cannot be thrown, respectively;
- **inNormalState** - flag that assumes the value true if the interactable is at rest and false otherwise;
- **inCarryState** - flag that assumes the value true if the interactable is being carried and false otherwise;
- **inThrowState** - flag that assumes the value true if the interactable was thrown and false otherwise;
- **inBalconyState** - flag that assumes the value true if the interactable is placed on a balcony and false otherwise;

When the player approaches a movable interactable, detects it, and presses the input *Interact*, the interactable is moved to the player's arms and from that moment on, regardless of where the player moves, the interactable will be moved with it. When the player carries the interactable the inNormalState flag becomes false and the inCarryState flag becomes true. Figure 4.47 illustrates a movable interactable being carried by the player.

Figure 4.47: Movable Interactable Being Carried By The Player

When the player carries an interactable, there are two actions they can do, the first is to press the *Use* input to throw the interactable forward, and the second action is to press the *Interact* input to drop the interactable. When the interactable is thrown, a force is added to the interactable's rigidbody, using the Unity function *AddForce()*, and the direction of that force corresponds to the player's forward vector. When dropping the interactable, two things can happen. If the interactable is dropped with a balcony in front, the interactable is placed on that balcony and changed to inBalconyState, if there is no balcony in front of it, the interactable drops to the ground and changes to inNormalState.
The movable interactable is further divided into several types, which inherit all their variables and functions, namely:

- Metal;
- Blade;
- Wood;
- Handle;
- Weapon;
- String.

4.6.2 Metal

The *Metal* is a movable interactable used in the forging process and has the following characteristics and variables:

- **normalModel** - model of the metal when in its normal state
- **heatedModel** - model of the metal when in its heated state
- **burnedModel** - model of the metal when in its burned state
- **heatProgress** - percentage from zero to 200 of how hot the metal is

- **heatSpeed** - speed at which the metal heats up
- **hammerSpeed** - speed at which metal is shaped
- **craftResult** - blade that results from the metal forging process

When the metal is placed in a furnace, the value of heatProgress increments a total of heatSpeed each frame, and when the heatProgress reaches the value of 100, the normal model is deactivated and the heated model is activated. When the metal is in its heated state, then the player can take it out and proceed with the forging process, however, if the player takes too long to take the metal out, the heatProgress will continue to increase and when it reaches 200, the metal passes to its burned state, which makes it completely useless. In figure 4.48, we can see the states normal on the left, heated on the center, and burnt on the right.

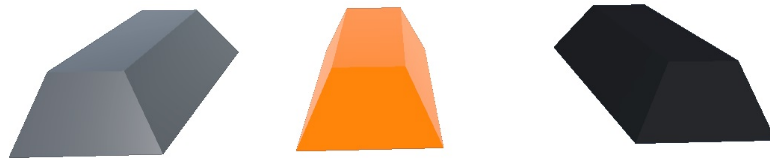

Figure 4.48: The Three States Of A Metal

Assuming the player has removed the metal in its heated state, the next step in the process is to place the metal in the anvil to shape it. When the metal is placed in the anvil, the player must hold the input *Use* to start and keep the process of shaping the metal. While the player maintains the shaping action, the shaping progress (stored in the anvil) is incremented by the hammerSpeed and when it reaches 100, the metal is destroyed and its craftResult is instantiated. Figure 4.49 illustrates the metal before and after the hammering process. On the left, we have the metal heated and placed in the anvil, and on the right, we have the craftResult of this metal.

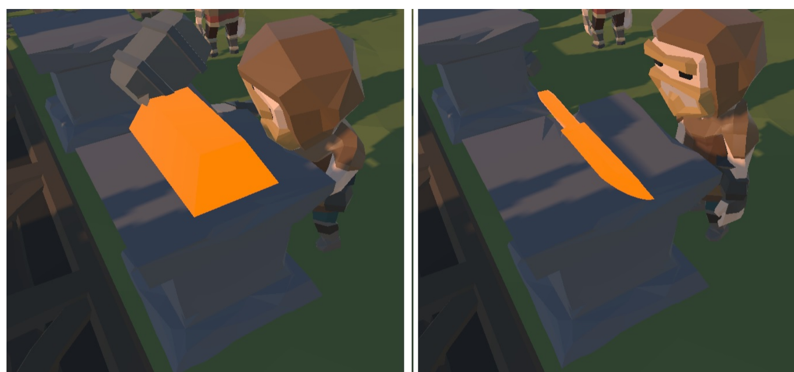

Figure 4.49: Metal Before And After The Hammering Process

A metal, when forged, results in a blade, but there are several blades made of the same metal. For instance, both the knife and the axe have their blade made of iron, but iron only has one craft result. To get around this problem, several replicas of the Iron metal were

created, and each one has a different blade as a craftResult, so, in a map where we want the player to forge axes, we use the iron that results in the axe blade.

4.6.3 Blade

The **Blade** is a movable interactable used in the forging process and has the following characteristics and variables:

- **ingredient** - metal needed to forge this blade;
- **listOfMergingPossibilities** - list of a tuple of type (ingredient, result) that contains all the interactables that the blade can be merged with, and the result of that merging;
- **heatedModel** - model of the blade when in its heated state;
- **quencheModel** - model of the blade when in its quenched state;
- **isQuenched** - flag that holds the value true or false depending on whether or not the blade is in the quenched state.

When heated metal is molded into the anvil, the result is a heated blade, which in this state is completely useless. To continue to be used in the forging process, the blade must be cooled down. At the level, there is always a cauldron with water where the player can cool the blades. When the player approaches this cauldron and presses the input *Use*, the weapon will be put in and out of the water, and before being taken out, the heatedModel is deactivated, the quenchedModel is activated and the isQuenched flag is set to true.

The blade is now ready to be joined with any of its merging possibilities. Figure 4.50 illustrates the blade before and after the quenching process. On the left, we have the heated blade and on the right, we have the quenched blade.

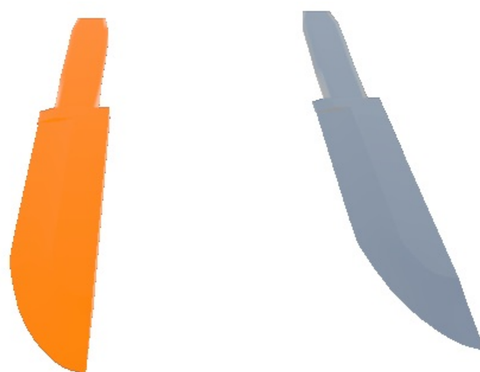

Figure 4.50: Blade Before And After The Quenching Process

4.6.4 Wood

The *Wood* is a movable interactable used in the forging process and has the following characteristics and variables:

- **shapeSpeed** - speed at which handle is shaped
- **craftResult** - handle that results from the wood shaping process

When the wood is placed in Saw, the player must keep the input *Use* to start and maintain the process of shaping the wood. While the player maintains the shaping action, the shaping progress (stored in the saw) is incremented by the shapeSpeed and when it reaches 100, the wood is destroyed and its craftResult is instantiated. Figure 4.51 illustrates the wood before and after the shaping process. On the left, we have the wood placed in the saw, and on the right, we have the craftResult of this wood.

Figure 4.51: Wood Before And After The Shaping Process

A wood, when shaped, results in a handle, but there are several handles made of the same wood. This same problem happens with metal, and to solve it we use the same solution we used in the 4.6.2 section.

4.6.5 Handle

The *Handle* is a movable interactable used in the forging process and has the following characteristics and variables:

- **ingredient** - wood needed to forge this handle;
- **listOfMergingPossibilities** - list of a tuple of type (ingredient, result) that contains all the interactables that the handle can be merged with, and the result of that merging;

From the moment it is instantiated, the handle is ready to be merged with any of its merging possibilities. Figure 4.52 illustrates an example of a handle.

Figure 4.52: Example Of An Handle

4.6.6 Weapon

The *Weapon* is a movable interactable used in the forging process and has the following characteristics and variables:

- **ingredients** - list of interactables needed to forge this weapon;
- **listOfMergingPossibilities** - list of a tuple of type (ingredient, result) that contains all interactables that the weapon can be merged with, and the result of that merging;

A weapon is the result of the merging between a blade and a handle and is usually the result of the forging process. In some situations, the weapon can be merged with other interactables like a string, to result in another version of the weapon with more details.

A weapon, from the moment it is instantiated, is ready to be delivered, without the need for intermediate processes. Figure 4.53 illustrates two versions of the same weapon. On the left, we have the Viking axe, resulting from the merging between a blade and a handle, and on the right, we have a refined version of the same weapon, resulting from the merging between the Viking axe and a string.

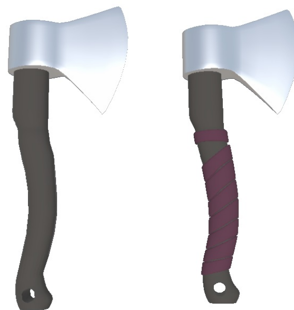

Figure 4.53: Example Of An Weapon

4.6.7 String

The *String* is a movable interactable with no additional properties or functions. The only purpose of this feature is to be merged to a handle or weapon, to result in a more refined item. Figure 4.54 illustrates an example of a string.

Figure 4.54: Example Of An String

4.6.8 Static Interactables

Static Interactables are interactables that cannot be moved by the player and physics, they are divided into 3 types:

- **Balcony** - interactable where the player can place other interactables;
- **MaterialSource** - interactable where the player can pull materials from, like metal and wood;
- **QuickAction** - interactable where the player performs a quick action, which cannot be interrupted.

4.6.8.1 Balcony

A balcony is an interactable where the player can place another interactable, or perform some kind of active or passive action. Active action is a type of action where the player must be near the balcony and hold the input *Use*. On the other hand, passive action is a type of action that does not require the presence of the player. Regardless of its type, the balcony has the following features and characteristics:

- **Interactable In Balcony** - reference to the interactable placed on the balcony. Assumes the value null if none exists;
- **Placement Point** - reference to the position and rotation that the interactable assumes when placed on the balcony;
- **CanPlaceInBalcony()** - function that checks if the player can place the interactable in the balcony;
- **CanRetrieveFromBalcony()** - function that checks if the player can remove the interactable from the balcony.

When the player approaches a balcony with an interactable in hand and uses the *Interact* input, it is checked if nothing has yet been placed on it, and if it is free, the interactable in hand is placed on the balcony. When the player approaches a balcony with an interactable placed there and uses the same input, it checks if the player's hands are free, and, if so, the interactable is moved from the balcony to the player's hands. The figure 4.55 illustrates a balcony with an interactable placed on it.

Figure 4.55: Interactable Placed In A Balcony

To handle passive and active actions, two types of balconies were created, *ActiveBalcony* and *PassiveBalcony*. These balconies work similarly, differing only in the need for the player to be present or not. These two types of balconies have a *Progress Bar* built through the *UI_Slider*, mentioned in the 4.2.3 section, which is used to show the player the progress of actions.

When the player approaches an *ActiveBalcony*, to use it he must first try to land an interactable on it, such as a metal, and then use the input *Use*. Every frame checks if the player is holding the input *Use*, and if this is true, the action continues, otherwise, the action stops. While the action is being performed, the player plays an animation that varies depending on the type of balcony. An example of an *ActiveBalcony* is the Anvil, in which the player can place a heated metal, and then hold the input *Use* to continue the shaping action. Figure 4.56 illustrates an example of an *ActiveBalcony*, the *Anvil*, where the metal is resting on the balcony and the action is progressing.

Figure 4.56: Anvil, a ActiveBalcony

When the player approaches a *PassiveBalcony*, to use it he just needs to place a compatible interactable on it. The operation is identical to *ActiveInteractable*, however, the action progresses independently of the player's behavior and also the player can stop and resume the action at any time. An example of a *PassiveBalcony* is the *Furnace*, in which the player can place metal and wait for it to heat up. While the metal heats up, the player can take any other action. Figure 4.57 illustrates an example of a *PassiveBalcony*, the *Furnace*, where the metal is resting on the balcony and the action is progressing.

Figure 4.57: Furnace, a PassiveBalcony

4.6.8.2 Material Source

For the player to carry out the forging process, materials are needed, which the player must take from the *MaterialSources*. This type of *static interactable* works quite simply,

and has the following features and variables:

- **material** - type of material that the player can remove;
- **RetrieveMaterial()** - function that instantiates a new material and places it in the player's hands.

When the player approaches a *MaterialSource*, we check if his hands are free, and if they are, then he can interact. When using the input *Interact*, the intended material is instantiated and moved into the player's hands. Figure 4.58 illustrates the example of a *MaterialSource*, in this case, for Iron.

Figure 4.58: Example of an MaterialSource

4.6.8.3 Quick Action Balcony

During the game, some are fast and cannot be interrupted, such as merging two components or cooling a blade. These types of actions are performed at *QuickAction Balconies*. These types of balconies differ a lot from each other and therefore each one needs a different behavior script. In the game, there are two *QuickAction Balconies*, the *Quencher*, and the *Table*.

The quencher is the interactable where the player can cool forged blades and when the player approaches it with a heated blade in hand and presses the *Use* input, the action begins. In this action, the player plays an animation and at the end of the animation, the heated blade model is deactivated and the base blade model is activated. The figure 4.59 illustrates the *Quencher* balcony, as well as the player performing the corresponding quick action.

Figure 4.59: Quenching Quick Action

The Table is the balcony where the player can merge blades, handles, strings, and weapons. To use this balcony to join the objects X and Y, the player must first place object X on the balcony and then, with object Y in hand, press the input *Use*. At that moment, it will be searched in the *listOfMergingPossibilities* (mentioned in the 4.6.1 section) of object X if there is any result for merging it with object Y, and if there is, the action starts. In this action, the player plays an animation and at the end of the animation, the two objects are destroyed and the result of the merge is instantiated. Figure 4.60 illustrates the *Table*, a *QuickAction Balcony*.

Figure 4.60: Table, an Quick Action Balcony

4.6.8.4 Delivery Point

There is also another balcony, which does not fit into any of the previous categories, the *Delivery Point*. This type of balcony does not play any action or animation. To use this balcony, the player must simply approach it and press the input *Interact*, and if the player has one of the orders of the screen list on his hands, then that order will be completed and the points will be increased. Figure 4.61 illustrates the *Delivery Point* present in every level.

Figure 4.61: Delivery Point

4.7 Level Specifics

The levels are mainly distinguished by their topology and by the Orders that the player has to forge, however, **some levels have some elements with specific behaviors**.

4.7.1 Wagon

At Level 2, the *DeliveryPoint* moves in rails, mimicking the behavior of a wagon. Three points were established in the scenario, one in the center of the tracks and two at the two ends. As soon as the level starts, after starting the timer, the execution of a *Coroutine* responsible for moving the Wagon begins.

As soon as *Coroutine* starts its execution, using the *Rigidbody* present in *DeliveryPoint*, we add a velocity to it that moves it from the center of the tracks to the far left. When the distance between the Wagon and the point on the far left is less than a predetermined value, the wagon movement is stopped by setting the speed of *Rigidbody* to zero and the *Coroutine* is paused for 2 seconds . At the end of these 2 seconds, we resume the *Coroutine* execution, place the car at the point on the far right and start moving it towards the point at the center of the tracks. When the distance between the Wagon and the point on the far right is less than a predetermined value, we stop the wagon movement again and the *Coroutine* is paused for another 2 seconds. This process is repeated until the level

ends. Figure 4.62 illustrates the behavior of the Wagon.

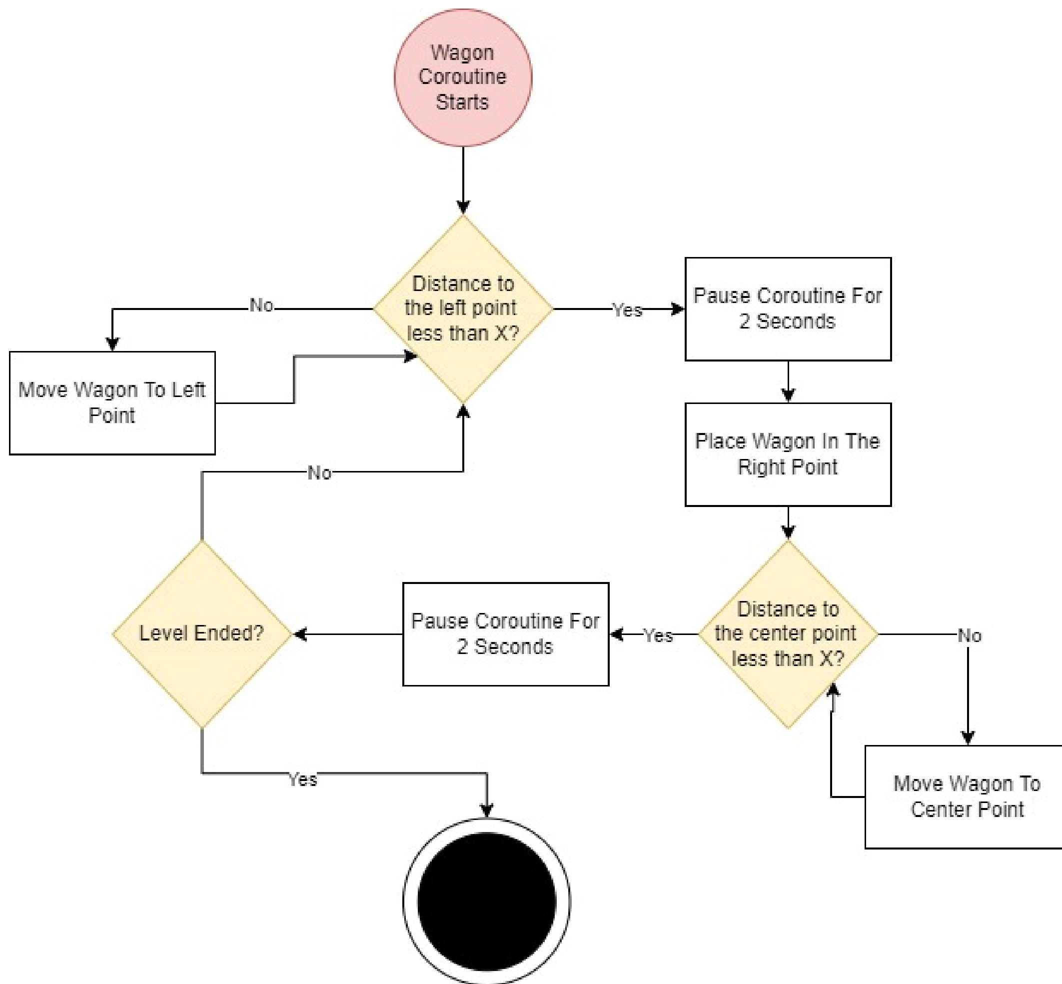

Figure 4.62: Wagon Behavior

The player needs to cross the tracks to collect materials and needs to approach the *DeliveryPoint* to deliver the Orders, however, if during this process the player collides with the wagon and the wagon is moving, the player is run over. When the player is run over, we ignore any of the gameplay inputs, and an animation plays. After 3 seconds, the player reappears in the same place where he started the level, and the inputs become available again. Figure 4.63 illustrates the Wagon, moving over the tracks to the point on the left.

Figure 4.63: Wagon, The Level 2 Delivery Point

4.7.2 Raft

At Level 3, there are two rafts and each player starts the game in one of them. Similar to the Wagon's behavior, six points were created in the level, and each of the rafts moves between 3 of those 6 points. Rafts are made of a model and a collider. Rafts do not have a *Rigibody* and therefore are moved by smoothly changing their position values. When the level starts, after starting the timer, the execution of a *Coroutine* responsible for moving the rafts starts.

As soon as the *Coroutine* starts its execution, using the *Lerp* function of the *Vector3* class, the new position for each of the rafts is linearly interpolated between the position of that raft and the left point. When the distance to the left point is less than a predetermined value, the *Coroutine* is paused for 4 seconds. At the end of the 4 seconds, the *Coroutine* is resumed and we move the Raft to the central point and after reaching that point, we wait again. Then we move to the point on the right where we repeat the process, and finally, we move back to the central point. This process is cyclically repeated until the level ends.

Players must stay in the Rafts for the entire level and exchange items between them by throwing items between the Rafts or when the two Rafts merge allowing players to cross. If players fall off the Rafts, a sound effect will play and the player will drown. Just like when the player is run over, wait 3 seconds and the player is replaced at the point where he started the level. Figure 4.64 illustrates the Rafts in Level 3.

Figure 4.64: Rafts in Level 3

Chapter 5

Tests

Testing is important not only after development, but also throughout the development process, so if it turns out that a mechanic has been poorly implemented, it is possible to redesign it before creating more dependencies, thus avoiding a more complicated refactoring process.

5.1 Preparing The Game For The Tests

The mechanics of *Overforged* are known to work due to the success of the game *Overcooked*. What is not yet known is whether **these same mechanics**, which work in the context of cooking, **have been properly transposed into the forging process**. Test results may lead to changes in the game concept, or just adjustments to speeds and values, or both. Although we cannot prepare the game to have its core changed, we can prepare it to have its values changed, within which:

- Character movement speed;
- Distance of the character dash;
- Force with which objects are thrown;
- Time required for hammering, sawing, and heating action;
- Level scores;
- Scores and times of orders.

Since the beginning of the development process, the good practice of putting all values that impact the gameplay of the game in the format of visible variables in the editor was taken into account. In this way, if we think that a value should be lower or higher, we simply change the value of the corresponding variable, without the need for any change in the code. Take as an example the speed of the playable characters, which was implemented in the variable format. If we want to change this speed, we change it in the corresponding field in the editor, represented in figure 5.1 with the name ”Movement Speed”.

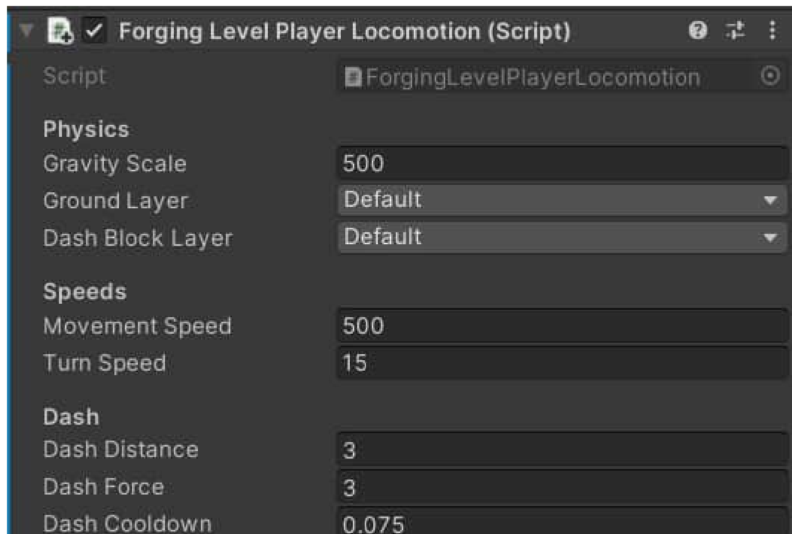

Figure 5.1: Values related to player locomotion represented in the editor

In addition to values, it may also be necessary to change the game models or images. **To avoid these changes bringing errors or problems, the models must be just visual components**, without any type of associated behavior, and without being referenced anywhere in the project.

In figure 5.2 the hierarchy of the weapon Knife is visible, in the hierarchy we can see that the knife model, corresponding to the object ”Knife” is a child object of the object ”Model”. The ”Model” object is an empty object, but at the code and implementation level, it is this object that acts as a model. The ”Knife” object is for visual purposes only and has nothing associated with it, which means that if it is deleted there will be no pointers or behavior errors.

Figure 5.2: Knife Weapon Hierarchy

All objects with models or images inserted in the game have been ranked in this way, and therefore they can all be swapped without problems. With this, the preparation of the game for the tests is done.

5.2 Planning The Tests

For the tests to be carried out, the participants will start with the game just opened, that is, on the screen with the message ”Press Start To Play”. From there, participants will be told to feel free to do whatever they want from the main menu and when they want to start a new game.

From this point on, **no help or guidance will be offered**, and players will be asked to complete the first level. When players complete the first level, they will be asked to play two more levels, so they can use all sorts of interactive objects and mechanics implemented in the game. According to the game rules, players will need at least one star to unlock the next level, so **if players don't get at least one star, some guidance and help will be offered**.

The tests will be carried out in pairs so that participants can play cooperatively, and **at the end** of testing the 3 levels, **they will be asked to fill in a questionnaire** where they can leave their feedback. The questionnaire used was based on **Whitton's questionnaire** that is described in their thesis ”An Investigation into the potential of collaborative computer game-based learning in Higher Education” (Whitton, 2007, p. 153).

This quiz is great because it focuses on evaluating the player's experience, but it's also a generic quiz, and some questions do not apply to all game genres. This means that some of the questions do not apply to the *Overforged* game, such as the question ”I was not interested in exploring all of the environment”, because in this game the player is limited to the scenario shown on the screen, there is no more to explore. The questions that were considered not applicable to the game or that were repetitive were removed from the questionnaire, reducing the number of questions from 42 to 23.

Some of the questions introduced in the Whitton questionnaire are placed in the negative and others in the positive. This can confuse the interpretation of the questions by the participants and therefore, the questionnaire was changed so that all questions were placed in the positive. An example of this change is in the question ”I did not find it easy to get started”, which was changed to ”I found it easy to get started”.

Whitton's questionnaire consists of **measuring the participants' engagement score**, through the use of a **Likert Scale** which was invented by the psychologist Rensis Likert (Wikipedia, 2022b). **This scale can measure different aspects and be presented in the most diverse forms and the most diverse quantities**. In the questionnaire, the Likert Scale will be presented with the following options:

- **Strongly Disagree**;
- **Disagree**;
- **Neutral**;
- **Agree**;
- **Strongly Agree**.

Figure 5.3 illustrates the Likert Scale applied to a question of the Nicola Whitton questionnaire.

Figure 5.3: Likert Scale applied to a question of Nicola Whitton questionaire.

Finally, **an extra question was added, where participants can write any additional feedback** that was not covered by the questions. **This last question is not mandatory** and the questionnaire can be submitted with it blank. The questionnaire was created using Google Forms, and the questions that compose it can be consulted in the appendix A.

5.3 Tests Results

The tests were carried out with students from the 1st year of the master's degree in "Digital Game Design and Development" from 2021/2022, consisting of 8 participants. In these tests, **it was possible to observe the behavior of each of the participants while they were playing** and to have, in addition to the questionnaire, more personal and specific feedback.

In addition to these tests, the game was made available for free download on *itch.io* (Itch.io, 2022). The game was downloaded by 10 players until now, from which 3 answered the questionnaire and left their feedback, totaling 11 answers.

The questionnaire mentioned in section 5.2 consists of 23 questions answered with the *Likert Scale* and an optional question. **For each of the 23 questions, there is a good answer and a bad answer**. To calculate the results of the questionnaire, **a score was assigned to the answers**, with the optimal answer being worth 4 points and the poor answer being worth 0 points. Take as an example the question "I found it easy to get started", in this question, the answers would be worth the following points:

- Strongly Disagree: 0 points;
- Disagree: 1 point;
- Neutral: 2 points;
- Agree: 3 points;
- Strongly Agree: 4 points.

That said, for each question, **the best result is obtained when the 11 answers point to the answer that is worth 4 points**, totaling 44 points, **and the worst result is obtained when the 11 answers point to the answer that is worth 0 points**, totaling 0 points. As 23 questions were implemented, the best result will be a total of 1012 points (23 questions x 44 points), and the worst result will be a total of 0 points (23 questions x 0 points).

The calculation of the points of each question is simple, observing the graph generated by google forms it is possible to see the number of people who selected each of the answers. Figure 5.4 illustrates the graph generated for the question ”I wanted to complete the activity”.

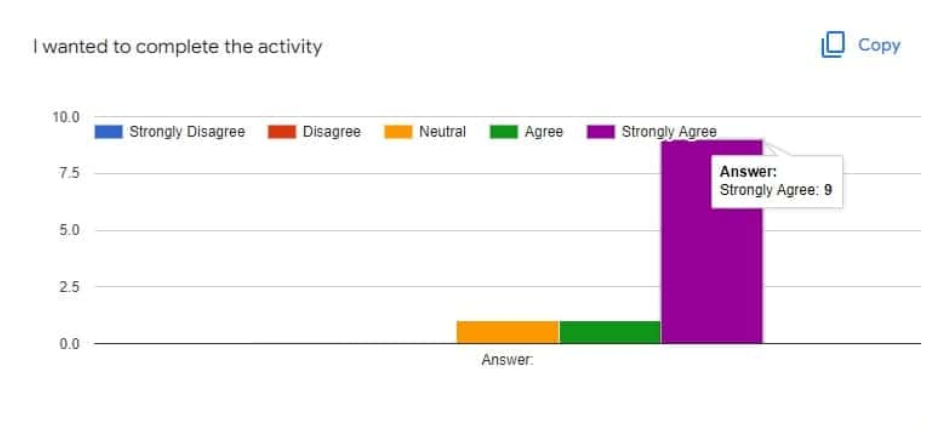

Figure 5.4: Example of the graph generated by the forms for one of the questions.

For the question ”I wanted to complete the activity”, 9 people chose the option worth 4 points, 1 person chose the option worth 3 points, and the remaining person chose the option worth 2 points. In this case, the total score for the question is:

$$(9 \times 4) + (1 \times 3) + (1 \times 2) = 36 + 3 + 2 = 41$$

This question scored 41 points out of a total of 44 points, which means that of the participants who answered the questionnaire, 93% wanted to complete the activity.

The same calculation was performed for the remaining questions and finally, the scores of the 23 questions were added up, obtaining a sum of 800 points in a total of 1012, **which corresponds to a satisfaction of 80%**. **Participants had more difficulty understanding the purpose of the game and how to get started (these being the responses with the lowest scores)**.

Finally, 7 of the participants answered the optional question, leaving some additional feedback. Of the 7 responses, 3 of them talk about the participants' appreciation of the game's graphic style, or about their enjoyment during the tests. The other 4 answers are suggestions, which suggest changes to the game, being these:

- Decrease the size of the game's UI, as it takes up a lot of game space in level 3;
- Increase the light emitted by items when the player approaches them;

- When starting a new game, do not open the world map, as you cannot choose any level other than the first one, so it makes more sense to go directly to level 1;
- It should be more explicit where to find the materials.

5.4 Tweaking Game

After analyzing the test results, **we will change the game to fix the problems mentioned** by the 4 suggestions made in the optional question, mentioned in the 5.3 section.

5.4.1 Size Of The Game's UI In Level 3

One of the participants suggested "Decrease the size of the game's UI, as it takes up a lot of game space in the water level", this water level being level 3.

At level 3, players are placed on rafts, and each player performs a set of tasks available on the raft they are on. **The player placed in the top raft has more difficulty in carrying out his tasks** because sometimes, during the movement of the raft, **the UI relative to the Orders obstructs the player's view**.

The size of the UI relative to Orders cannot be reduced, as it would be too small. Moving the rafts is also not a solution as in that case the UI of the score and time would obstruct the player's view of the bottom raft. **The solution is then to move the camera away from the players**, to put more content in the field of observation of the players. Figure 5.5 illustrates the new camera view in level 3, fixing the UI overlapping issue.

Figure 5.5: New camera view in level 3, fixing the UI overlapping issue.

5.4.2 Increase Light Emitted By Items

When the player approaches an object, the object's material changes to emit a certain color, simulating a light source. This emission process was mentioned in the 4.6 section, and as described, the color emitted by the material has 4 channels, 3 for color and 1 for

intensity. The 3 color channels are the Red, Green, and Blue channels, and the intensity channel corresponds to how bright the color is. **To increase the light emitted by items, simply increase the value of the intensity of the emitted color**.

5.4.3 Loading the First Level Instead Of The World Map

In the main menu, when the player starts a new game, the World Map is loaded, and **the player does not have any level to choose from, he can just press Enter to open the first level**, or use the pause menu to return to Main Menu. This is confusing for the player, as it makes no sense to place the player on the world map if he cannot choose between the various levels.

To solve this problem, in the Main Menu, **when selecting the "Start Game" button, instead of loading the Main Menu, Level 1 is loaded directly**, using the Loading Manager described in the 4.2.7.

5.4.4 Making The Process of Finding Materials More Explicit

Some participants had difficulty finding the materials. Although the player's need to look for the source of materials is one of the goals of the game, **it is not intended to frustrate the player**. To balance these two points, **a screen was added in each of the tutorials with a direct illustration of each of the balconies with the materials, but the location on the map is not revealed**. Figure 5.6 illustrates an example of one of the added panels.

Figure 5.6: Example of one of the added panels.

Of course, **these panels were only placed for materials that appear for the first time,** that is, at level 1 the player interacts with iron for the first time, and therefore, in the level 1 tutorial, there is a concrete illustration of the object that contains the iron. At level 2, the iron illustration will no longer be placed as it is nothing new.

It is considered that this is not a big problem because after the player plays the level for the first time, in the following times he already knows

where the materials are.

Chapter 6

Conclusions And Future Work

The most relevant conclusions of this project as well as the prospects for future work are presented in this chapter.

6.1 Conclusions

The making of this project allowed interaction with a wide range of technical components that make up the process of developing digital games, such as modeling, animation, programming, level design, and sound. This project allowed an increase in the mastery of the Unity game engine and the *C#* programming language, as well as the *Blender* tool.

Video game development is not a technically easy task, because the developer needs to program a real-time experience with high-quality graphics that run smoothly and without delays. Game engines help to overcome these problems, making development more accessible.

Making a game is increasingly easier and at the same time, more difficult. As mentioned before, with the emergence of new software and tools, the technical process becomes easier, and therefore, more and more games are made available on the market, which makes it difficult for new games to stand out.

In this project, a game was developed whose mechanics are based on an existing game, but which is still considered innovative, as there is no fast-paced and cooperative game with the theme of forging.

6.2 Future Work

The *Overforged* has not yet come to an end. The core of the game is developed, and the future of this game lies in improving existing content and adding new content.

The first step of future work is the implementation of new levels and new game modes, to make this game complete and worth the player's time. In the new game modes, players will not only be able to cooperate, but also compete against each other. The game should then be tested to see if the new content is enjoyable and works well.

Next, the co-op system will be expanded to support up to 4 players. This process won't be too complicated as the game has already been developed with this upgrade in mind.

When everything is guaranteed to be functional, a network system will be implemented, so that, in addition to locally, players can play with each other over the internet. The implementation of this system will not be easy, as ensuring that a fast-paced

game runs smoothly over the internet requires movement prediction algorithms and good management of the data that is exchanged between players.

Still, to increase the level of immersion and to please the most competitive players, a trophy/achievement system would be implemented similar to the one already present in current consoles, the player would receive a trophy for performing certain actions.

References

Flame, W. T. (2020). *Types of quenching process for blacsmithing.* Retrieved 09 January 2022, from https://workingtheflame.com/blacksmith-quenching-guide/ 4

IndustrialHeating. (2011). *History of blacksmithing.* Retrieved 09 January 2022, from https://www.industrialheating.com/articles/90136-history-of-blacksmithing 3

Itch.io. (2022). *Overforged by eliseu batista.* Retrieved 10 June 2022, from https://eliseubatista99.itch.io/overforged 102

Metacritic. (2018). *Overcooked 2 for pc reviews - metacritic.* Retrieved 09 January 2022, from https://www.metacritic.com/game/pc/overcooked!-2 5

Microsoft. (2022a). *Binaryformatter class - c# programming guide.* Retrieved 24 February 2022, from https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter?view=net-6.0 54

Microsoft. (2022b). *Delegates c# programming guide.* Retrieved 24 February 2022, from https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/ 47

Microsoft. (2022c). *Surrogateselector class - c# programming guide.* Retrieved 24 February 2022, from https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.surrogateselector?view=net-6.0 54

Rogers, S. (2010). *Level up! the guide to great video game design.* Jonh Willey and Sons. Retrieved from https://www.amazon.com/Level-Guide-Great-Video-Design/dp/1118877160 7, 8, 9, 10, 12, 13, 14, 15, 17, 20, 21, 23, 27, 34, 35

Synty. (2022a). *Polygon mini - fantasy characters pack.* Retrieved 15 January 2022, from https://syntystore.com/products/polygon-mini-fantasy-characters-pack 42

Synty. (2022b). *Polygon mini - fantasy pack.* Retrieved 26 January 2022, from https://syntystore.com/products/polygon-mini-fantasy-pack 63, 64

Synty. (2022c). *Polygon mini - particle fx pack.* Retrieved 26 January 2022, from https://syntystore.com/products/polygon-particle-fx-pack 81

Team17. (2018). *Overcooked 2 - team 17.* Retrieved 09 January 2022, from https://www.team17.com/games/overcooked-2/ 4

TFGUSA. (2020). *Metal forging processes, methods, and applications.* Retrieved 09 January 2022, from https://www.tfgusa.com/metal-forging-processes-methods/ 3

Thompson, W. (2020). *Overcooked 2 pc review - hookedgamers.* Retrieved 09 January 2022, from https://www.hookedgamers.com/pc/overcooked_2/review/article-2168.html 5

Unity. (2022a). *Canvas | unity ui.* Retrieved 26 January 2022, from https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UICanvas.html 45

Unity. (2022b). *High definition render pipeline overview.* Retrieved 06 May 2022, from https://docs.unity3d.com/Packages/com.unity.render-pipelines.high

-definition@13.1/manual/index.html 37

Unity. (2022c). *Unity - manual: Navmesh agent*. Retrieved 26 January 2022, from https://docs.unity3d.com/Manual/class-NavMeshAgent.html 65

Unity. (2022d). *Unity - manual: Scriptableobject*. Retrieved 24 February 2022, from https://docs.unity3d.com/Manual/class-ScriptableObject.html 55

Unity. (2022e). *Unity - scripting api: Physics.raycast*. Retrieved 26 January 2022, from https://docs.unity3d.com/ScriptReference/Physics.Raycast.html 80

Unity. (2022f). *Universal render pipeline overview*. Retrieved 06 May 2022, from https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@13.0/manual/index.html 37

Whitton, N. (2007). *An investigation into the potential of collaborative computer game-based learning in higher education*. Retrieved 06 May 2022, from http://researchrepository.napier.ac.uk/id/eprint/4281 101

Wikipedia. (2022a). *Forged in fire - wikipedia*. Retrieved 25 April 2022, from https://en.wikipedia.org/wiki/Forged_in_Fire 6

Wikipedia. (2022b). *Likert scale - wikipedia*. Retrieved 11 June 2022, from https://en.wikipedia.org/wiki/Likert_scale 101

Appendix A

Game Engagement Questionaire

This appendix consists of the 24 questions asked in the Game Engagement Questionaire answered by the participants, referred to in the 5.2 section. The questions asked were the following:

1. I wanted to complete the activity;
2. I wanted to explore all the options available to me;
3. I cared about how the activity ended;
4. I knew what i had to do to complete the activity;
5. The goal of the activity was clear;
6. The instructions were clear;
7. I found it easy to get started;
8. I felt that i could achieve the goal of the activity;
9. I had a fair chance of completing the activity successfully;
10. I found the activity frustrating;
11. The activity was challenging;
12. The types of task were too limited;
13. It was clear what i could and couldn't do;
14. The activity was too complex;
15. I found the activity satisfying;
16. I felt absorbed in the activity;
17. I felt that time passed quickly;
18. I felt excited during the activity;
19. I found the activity boring;
20. The activity was aesthetically pleasing;
21. The activity was pointless;
22. The feedback i was given was useful;

23. The activity was worthwhile;

24. Additional feedback.

Questions 1 to 23 are answered using the *Likert Scale* referred to in the 5.2 section, and question 24 is an open-ended question.