UNIVERSIDADE
BEIRA INTERIOR

Development of an Educational Game: Math-Masters

João Abel Bento Tátá Marques

Relatório de projeto para obtenção do Grau de Mestre em
Design e Desenvolvimento de Jogos Digitais
(2º ciclo de estudos)

Orientador: Doutor Frutuoso Gomes Mendes da Silva

Covilhã, Junho 2023

Declaração de Integridade

Eu, João Abel Bento Tátá Marques, que abaixo assino, estudante com o número de inscrição M11645 de Design e Devenvolvimento de Jogos Digitais da 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 referência de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 8/06/2023

Acknowledgements

Para começar, agradeço primeiramente aos meus pais por me apoiarem da melhor maneira possível, tratando-me como seu filho. Permitiram-me seguir a carreira que mais gosto e deram-me condições para me permitir desenvolver tanto as minhas competências adultas como as minhas competências profissionais. Nunca me restringiram na hora de tomar decisões, dando-me a liberdade necessária para progredir no meu próprio ritmo. Agradeço também aos meus irmãos, cada um fez mestrado e doutorado e deu um pouco de inspiração para seguir nessa próxima etapa acadêmica.

Agradecimentos especiais aos meus amigos mais próximos, Tibbe Höppener e Rodrigo Gewehr de Araujo. Passamos incontáveis momentos juntos, nós os três combinando nosso poder anti-social jogando videogame, jogando cartas. Eles auxiliaram me quando eu estava iniciando o curso de Engenharia da Computação, pois eu lutava para interiorizar o jeito do programador e como é o trabalho de um programador profissional. Fazemos projetos divertidos juntos. Estamos juntos há muito tempo e torço por mais.

Um agradecimento dedicado a Eva Sindicq. Ela tem me ajudado na transição para um adulto totalmente crescido, ouvindo minhas dificuldades sociais e dando conselhos importantes para a vida sobre como o mundo funciona. Descontraímos juntos do trabalho (e da vida em geral) e nos divertimos fazendo nossas atividades favoritas. Eu valorizo nosso relacionamento e espero continuar nosso apoio mútuo.

Quero agradecer imensamente aos meus colegas/companheiros de casa Vinicius Espinosa e João Mayer. Esses últimos dois anos seriam impossíveis sem eles. Para mim éramos mais que colegas, éramos companheiros trabalhando juntos para alcançar o objetivo de fazer o que gostamos e aprender com os desafios que enfrentamos. Participamos de GameJams juntos, fizemos projetos semestrais juntos. Além do trabalho,

mudar para o mesmo apartamento que eles foi a melhor decisão que tomei no ano passado, ao contrário de ficar sozinha em um quarto como fiz no ano anterior. Fora do horário de trabalho, socializamos juntos, me dando uma verificação ocasional da realidade e apontando minhas falhas, ou tirando sarro de nossas diferenças culturais.

Devo agradecer ao João Ferreira por me aceitar como colega de projeto. Acredito que sem um Designer, eu não teria um projeto para começar.

Trabalho desenvolvido no "Instituto de Telecomunicações" sob orientação do Prof. Frutuoso Silva. O seu papel como Professor de Programação neste Mestrado é o que me permitiu manter a programação ao longo destes últimos dois anos, já que este curso é sobre muitas outras áreas. Com ele pude falar sobre minhas dificuldades e descobertas como programador especializado em Unity, e ele me ensinou muitas lições importantes.

Agradeço a todos pela ajuda ao longo desta jornada. Que o futuro reserve mais.

Resumo Alargado

Desenvolvimento de *Math-Masters* - jogo com carácter didático e sério para aprendizagem da multiplicação básica. Para além de educacional, Math-Masters é um jogo deck-building roguelite na perspetiva de terceira pessoa, onde o jogador percorre trilhos com inimigos e tem de os derrotar no mundo medieval de fantasia. A principal inspiração de Math-Masters é *Fun Math Facts: Games for Kids*, desenvolvido por Speedymind llc, onde o jogador tem de se defender de monstros a ir na sua direção através da matemática. O utilizador possui um conjunto de cartas que permitem resolver os problemas matemáticos e dessa forma eliminar os inimigos e progredir no jogo.

A programação do jogo foi realizada por mim, a partir do design proposto pelo colega João Ferreira que tambem providenciou os modelos dos objetos, arte e o conteúdo do jogo.

Foi utilizado o motor de jogo Unity, com a integração da base de dados online PlayFab. Por isso o jogo requer internet para que o jogador se possa registar na base de dados e poder salvar o seu progresso.

Quanto à metodologia, este documento consiste em mostrar como foi feita a implementação das mecânicas do jogo e como está organizado.

O jogo está disponível para ser testado no Itch.io apartir do link seguinte: `https://thejokelion.itch.io/math-masters`

Palavras-chave

Jogo Educacional, Jogos Sérios, Jogo de Matemática, Unity

Abstract

Development of *Math-Masters* - educational/serious game for learning of the basic multiplication in mathematics. Beyond being educational, Math-Masers is a deck-building roguelike game in the third perspective, where the player runs through tracks of enemies and must defeat them, in a fantasy medieval world setting. The main inspiration for Math-Masters is *Fun Math Facts: Games for Kids*, developed by Speedmind llc, where the player must defend itself from incoming monsters heading their way through math.

The programming of the game was done by me, according to the design proposed by my colleague João Ferreira who also provided the models of the objects, art and game content.

The game engine Unity was used, with the integration of an online database from PlayFab.

About the Methodology, this document consists in showing how the game mechanics were implemented and how it is organized.

The game is available to be tested in Itch.io through the following link: `https://thejokelion.itch.io/math-masters`

Keywords

Educational Games, Serious Games, Math Games, Unity

Contents

List of Figures

Chapter 1

Introduction

This project was made during the second year of the Master's degree in Game Design and Development, and it surrounds the development of the serious/educational game Math-Masters.

1.1 Scope of the Project

Usually the programming of a game involves a team of programmers, but in this case the team is only composed of one member. A game created by one programmer does not necessarily mean simple and small. It can be a great game having in consideration the technical quality, the creativity and time management. For this I give the example of Toby Fox and the hit game he developed Undertale [1]. Undertale is a 2D roleplaying game. Developed in two and half years, with Toby Fox as the programmer (and writer) and Temmie Chang as the artist and character designer. This game is considered to be one of the greatest of all time. This shows that it is indeed possible to produce games with few programmers, although taking longer to develop.

This project is aimed to design and develop a point-and-click educational game for math, where the player has a deck of cards that can be used to solve math problems and destroy enemies. This document shows an overview of the approaches and options chosen during its development

1.2 Motivation

As a gamer, it has been a lifelong dream to develop videogames. There are many areas in videogame production, and as someone who's only been programming up to this point, I was uncertain as to what was my vocation in videogame production since I had an interest in many of those areas besides programming. During the first year of the Master's Degree, I tried many areas and reached the conclusion that programming is indeed my vocation and what I need to continue specializing and gaining more experience. This project is a testament of my newfound skills and ability to learn more so as to overcome obstacles and implement different/more complex features. It aims to develop a point-and-click educational game for math.

1.3 Objectives and Methodology

The goal of this project is to develop an online serious / educational game attached to genres to make it more fun and engaging. Focusing on the educational part of this game, where the player needs to utilize their math solving skills to be able to progress through the game.

This game will be developed in the Unity[2] game engine as it is a good base to attempt at developing new and different features without encountering a high difficulty bar on the get-go. This project does not intend to be ambitious because its programming was not developed by a team, and instead by a single programmer.

This game was developed according to the design proposals and utilizing the 3D assets and 2D assets provided by my colleague *João Ferreira* who co-developed the game by performing the role of Designer and Artist. This report will only discuss the programming side of Math-Masters.

1.4 Document Organization

This document is organized with the following chapters:

1. **Introduction** - Introduce the situation and reasons surrounding this project.

2. **Related Works** - Present existing games from the videogame industry that some-

what impacted the objectives and main goals from this project, either visually or functionally.

3. **Technologies and Tools Used** - Show what tools and technologies assisted or were used in the making of this project, with some comparison to alternatives.

4. **Design and Development of Math-Masters** - Describe how the game's features were implemented and how they interact with one another. Also how those features enable a better Design handling of the content.

5. **Conclusion and Future Work** - State how was my experience developing this game, and what is still left to accomplish.

Chapter 2

Related Works

2.1 Serious / Educational Games

Serious / Educational games are games meant to teach players about topics or help acquire skills. Utilizing interactive videogame elements, such as challenges, rewards, progress tracking and problem-solving activities, players will be invested into the gameplay and gladly learn what is intended. Design of an educational game requires more steps ([3]) than the traditional game, namely the inclusion of content to be learned (i.e., learning mechanisms [4]). On the subject of math, games meant for a targeted audience of children learning the multiplication operator require simplicity to not overwhelm the player, nor cause boredom for what is meant to be educational.

The two following games are the serious / educational games that showed the most features, in line with our objective. In comparison, most games we researched on had no combat elements and were more focused on different ways to display operations on screen. In other words, those games behave like simulated interactive classes rather than stimulating the players through gamification.

2.1.1 Fun Math Facts: Games for Kids

Fun Math Facts: Games for Kids [5] is about teaching math through challenges, disguised as combat. As shown in the figure 2.1, the player is represented by an avatar facing an approaching enemy. The enemy is moving towards the player, contacting with

the player results in ending the challenge as defeat. This is an implicit way of putting a timer as a way to challenge the player.

Figure 2.1: Screenshot of the gameplay when the player is prompted with a question

The only way to defeat the enemy is by selecting the correct answer to the problem, where the player's avatar will attack the enemy and eliminate it, as shown in the figure 2.2. Answering incorrectly will penalise the player by stunning the avatar and not allowing to attempt at attacking again for a short duration of time.

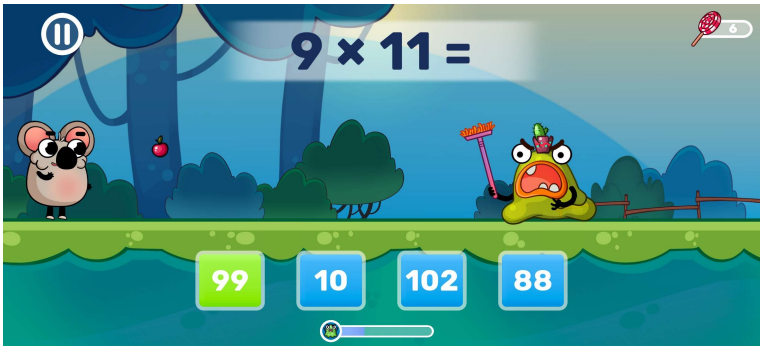


Figure 2.2: Screenshot of the gameplay when the player answers the question correctly

This game is divided into chapters displayed on a path traced on a map, as shown in the figure 2.3. Each chapter has a number of enemies to be defeated in order to progress. The game scores the player's completion of each chapter according to the amount of failed answers given during the challenge. Moreover, some chapters have a boss enemy included. These bosses are not taken down with one attack, and their movement is unimpeded when hit, meaning they get closer and closer to the avatar until defeated.

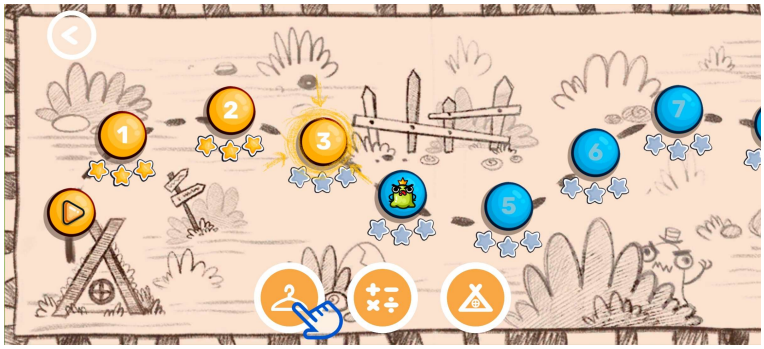


Figure 2.3: Screenshot of the gameplay map

This game has a way to personalize the player's avatar through changes in visuals, such as clothing, as show in the figure 2.4. Some enemies will drop a type of coin upon defeat, meant to be spent at the cosmetic store. This is a way to reward the player by playing more and according to the objective of the game.

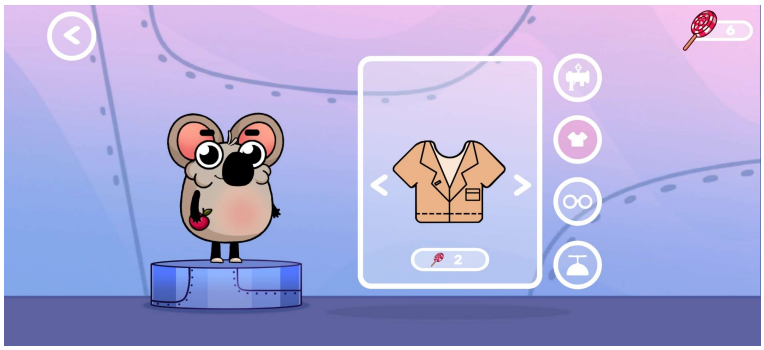


Figure 2.4: Screenshot of the gameplay cosmetic store

The pause menu of this game is simple, displaying very few buttons as shown in the figure 2.5 with only the base necessities - resume, restart and return (with mute/unmute buttons for sounds effects and music).

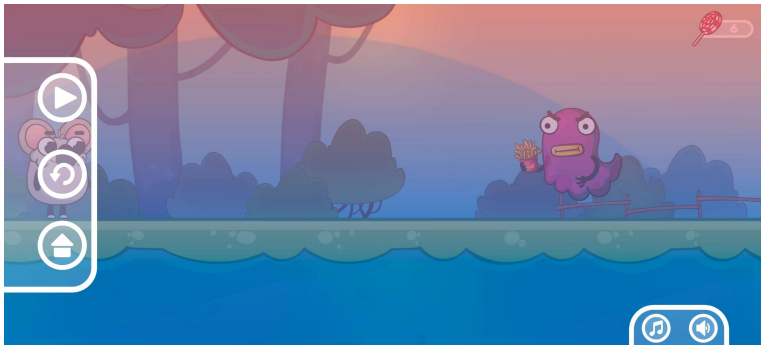


Figure 2.5: Screenshot of the gameplay pause menu

This game was also made on the Unity Engine [2].

2.1.2 Prodigy Math

Prodigy Math [6] is an educational and exploration game with the intention of teaching several areas of mathematics. The main gameplay feature of this game is turn based combat with stylized, simple and appealing to children aesthetics, as shown in the figure 2.6.

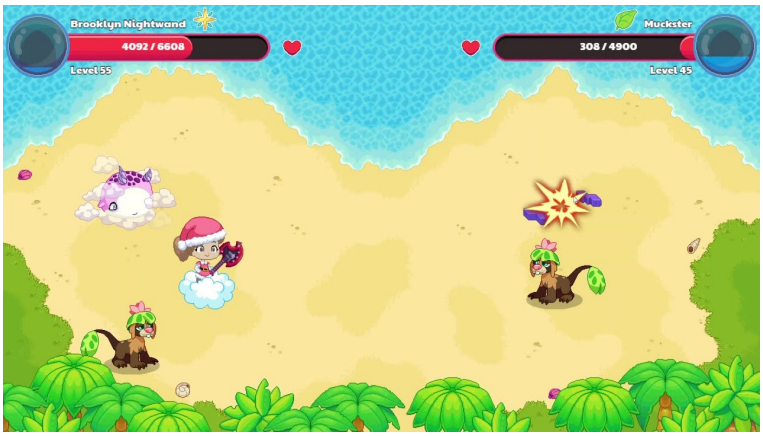


Figure 2.6: Example of a combat situation[1]

During combat, the way the player attacks the opponents is by answering a mathematics question, as shown in the figure 2.7. Answering correctly will result in an attack, answering incorrectly will result in missing the attack, allowing the enemy to fight back unimpeded. These questions can be of multiple-choice, gap filling or picking an image. However, to answer the question is used another screen, which breaks the gameplay.

[1] Image source

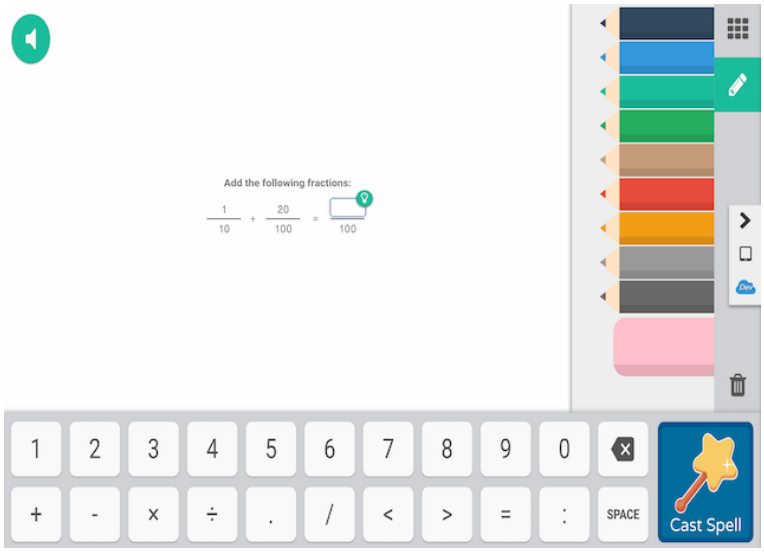


Figure 2.7: Example of a math question[2]

Players need to be registered with an account in order to play, so that their progress is saved online. For this the game has a login screen after booting up the game, as seen in the figure 2.8. A feature this game has is the ability for parents to check on their children's account progress.

Figure 2.8: Login screen[3]

This game also provides customization of the avatar's visuals, as shown in the figure 2.9.

[2]Image source
[3]Image source

Figure 2.9: Customization menu[4]

For the exploration aspect of this game, possesses a large and intricate map with a hub at the center and several areas with different themes and challenges for the players to face, as shown in the 2.10

Figure 2.10: World map[5]

This game is meant for an older targeted audience than our objective, since this game is more complex in both features and mathematical questions than the previous educational game.

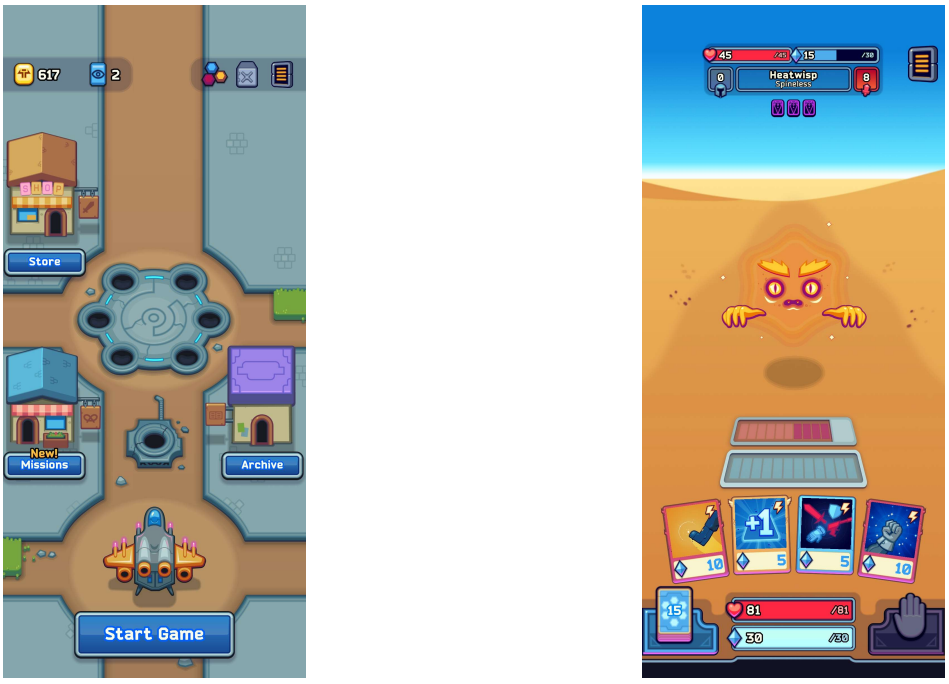
[4]Image source
[5]Image source

2.2 Inspiration Games

2.2.1 Void Tyrant: Eyes of Chronos

Void Tyrant: Eyes of Chronos [7] is a rogue-lite turn based deck-building game. Before the player starts a run, they have a hub, as shown in the figure 2.11a, where they can upgrade their equipment, start new missions and check the progress from previously made runs.

During the run, encountering an enemy starts combat, as shown in the figure 2.11b. To attack an enemy, the player has to play cards with numbers in order to have a total number greater than the opponent without overflowing (in similar style to the card game blackjack). A successful attack deals damage, failure results in taking damage. As staple in any rogue-lite/like game, if the player's health points reach zero, the run is over and the player has to start a new run.



(a) Hub **(b)** Combat

Figure 2.11: Screenshots of gameplay

In addition, the player has a separate hand of cards called spells. These spells have unique properties that influence many aspects of the combat, as shown in the figure

2.12a. They can, for example, regenerate player health, directly attack the opponent, debuff the opponent such a way that they can't attack.

The player holds a deck of spells, as seen in the figure 2.12b. For the duration of the run, the player has the option to add, remove, modify and duplicate cards of their deck.

(a) Example of a Spell card

(b) Deck of spell cards

Figure 2.12: Screenshots of gameplay

2.2.2 Slay the Spire

Slay the Spire [8] is a rogue-like turn based deck-building game. This game is what popularized the combination of the rogue-like and deck-building. The points to take inspiration from this game are similar to the previous game, namely the turn based mechanics, as shown in the figure 2.13, selecting cards to attempt at attacking the opponents, the ability to modify the deck of spells during the run (removing, adding, duplicating, modifying cards).

Figure 2.13: Screenshot of the gameplay combat

2.2.3 TUNIC

TUNIC [9] is what inspired the visuals of Math-Masters the most. The player's avatar is the stereotypical adventurer from stylised fantasy, short sized, holding a small sword and shield. Through a third-person perspective, the player explores the world, as shown in the figure 2.14.

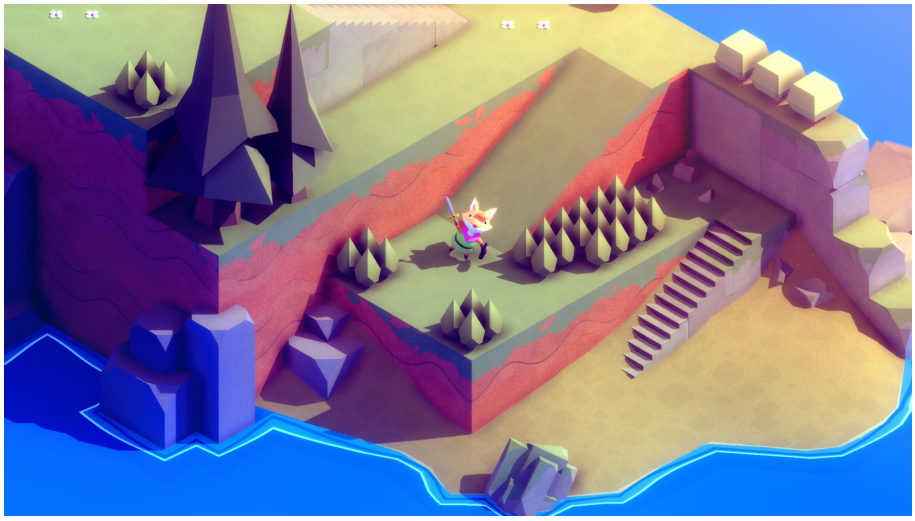


Figure 2.14: The main character from TUNIC.[6]

The 3D assets in this game are low poly and stylised. The enemies are visually cartoonish, lowering any fear the player may have of facing them, as seen in the figure 2.15. The exploration and adventure theme are important elements we intend to have in our game.

[6]Image source
[7]Image source

Figure 2.15: The combat in TUNIC.[7]

Chapter 3

Technologies and Tools Used

3.1 Unity

Unity[2] is a cross-platform game engine. This engine provides a multitude of tools for development and design of 2D or 3D games. There are various plugins, frameworks, and libraries available that enhance the engine's capabilities and provide additional functionality. These extensions can help developers in different aspects of game development, such as asset management, visual effects, audio, user interface, and more. These extensions can be either official or made by the community, the latter can be freely available or be paid content.

Community made content can be publicly accessed through the Unity Asset Store [10], through a web browser. Like the extensions, assets available in this store can be either free or paid, as shown in the figure 3.1. There are entries that allow free use under the condition they are not used for monetised games, due to the lack of licensing.

Maintained by a company named after this product, they focus solely on updating the Unity engine and providing customer support according to the several paid tiers they have available.

As it stands, Unity is one of the two most popular game engines in the gaming industry, the other one being Unreal Engine, which I will compare the two of them after the following point.

Unreal Engine [11]: Produced by Epic, a company that not only develops this game engine, but also maintains a cloud gaming platform called Epic Games Store [12], where they sell not only the game they develop and have Unreal Engine available for

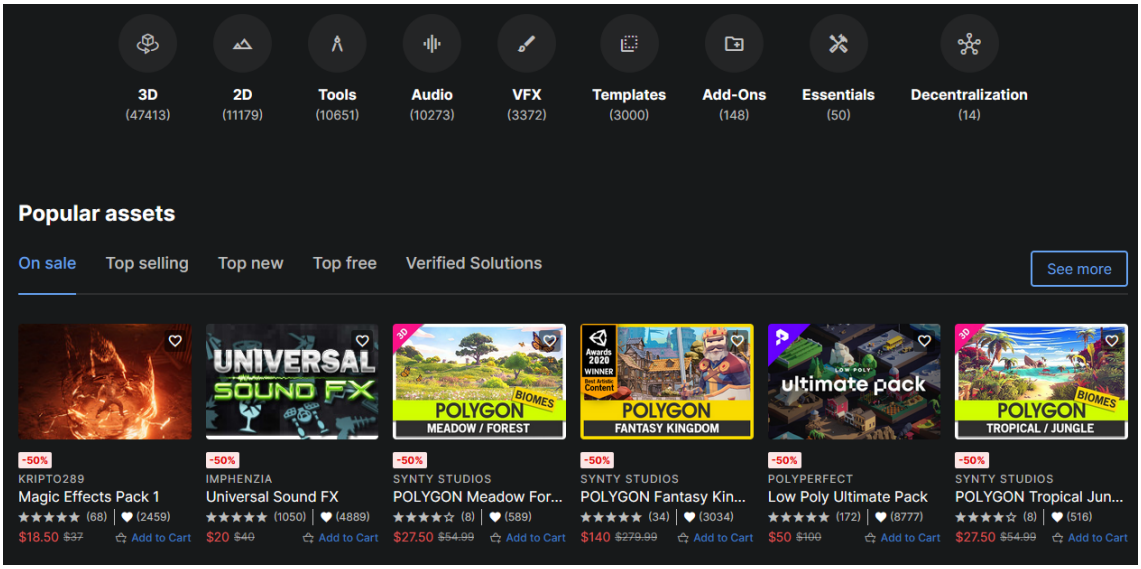


Figure 3.1: Screenshot of the website

installing, but also publish games from other studios.

Comparison between Unity and Unreal Engine:

- Unity has subscription models with different tiers according to the needs of the developers, providing a free version of unity as long as the games produced do not overtake an established limit of 100,000$ of annual revenue. In contrast, Epic's way to monetise Unreal Engine is through royalties. They too offer a free version until the produced games pass a revenue threshold of $1 million in gross revenue.

- For the programming language, Unity's C# is simpler and has more modern syntax, while the C++ from Unreal Engine is more complex with lower-level control and direct memory manipulation. On the other hand, C++ forces programmers to manually allocate and deallocate memory throughout the code running, according to the objects being used, while C# regularly and automatically cleans any unused memory and provides memory when needed.

- Unreal Engine's bigger selling point is the degree of realistic graphics it has by default. On the other hand, Unity is bare-bone in these terms by default, but this can be solved by integrating packages meant to enhance graphics or other needed properties.

- As for the operating systems, both function in the currently available computer OS Windows, macOS and Linux. For platforms outside of computers, they support iOS, Android, WebGL, PlayStation 4 and 5, Xbox One and Series X or S, Nintendo Switch and Augmented/Virtual Reality platforms.

The conclusion from the reasons above is that Unity is simpler and has a bigger community (both in assets and tutorials) which allows a much smoother start and the ability to learn both with help and on my own. On the other hand, Unreal Engine is overwhelming in the number of tools and methods available. In addition, Unreal's high difficulty learning curve in programming paired with the lack of free accessible tutorials (majority are behind paywalls) made me specialize in the Unity Engine. Through that conclusion that I picked the Unity game Engine for developing this project.

3.2 PlayFab

PlayFab [13] is a backend platform built for databases, performing the server-side of online games and potentially other services. The PlayFab API is made by Microsoft.

Microsoft: a multinational technology company that develops and sells a wide range of software, hardware, and services. Known for its flagship product Windows operating system, its mission is to provide innovative and accessible tools for everyone to use in their work or leisure needs.

If implemented into a game, players have to register to PlayFab through an account which can only be made through API calls. There is an alternative use that doesn't require players to make accounts, where PlayFab registers the computers used. This last method is unorganized and does not allow more than one unique player per device.

By being logged in, players will have their progress, leaderboards, inventories and any other relevant information stored in the databases provided by PlayFab.

PlayFab allows free creation and use of databases on up to 10 games, each having fewer than one hundred thousand unique players, after which it will require paying either through "Pay-as-you-go" or through monthly subscription. The more expensive the monthly plan, the better the customer support they will provide.

PlayFab has an optional integration package for Unity - called Unity3d PlayFab SDK [14] - making development of the API more intuitive and faster, showing information such as player data (more on that in chapter 4). It prompts the developer with a new window inside Unity asking to create a database for the game in the PlayFab website, pasting the database's encrypted code into unity for a safe and automatic connection to the API. This optional window also displays a hyperlink to open the database through a browser in one click named Game Manager, as shown in the figure 3.2.

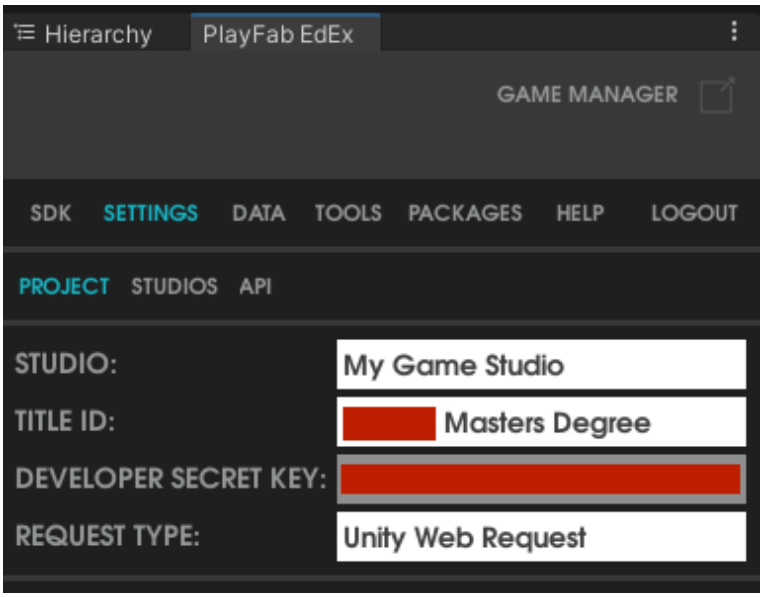


Figure 3.2: Screenshot of the website displaying Arduino instead of C# in a code block

When a player logs onto the game, PlayFab automatically associates the player's unique identifier onto the on-going game session. What this results is in subsequent requests made by that player do not need to specify the player's unique identifier, because PlayFab can retrieve it from the player's session. In other words, once the player enters the game, PlayFab will internally remember who entered without the developer having to remind it. This is known as the "authenticated player's context" in PlayFab.

This property streamlines user management and security, resulting in no need for saving the authentication and providing it to PlayFab whenever a database request is made.

A downside of using PlayFab is that there isn't much community content surrounding it. This results in few and far between tutorials and guides. Searching for terms on

browsers yields results that redirect to PlayFab's documentation, bare of explanations on how it works nor how to use it. The solution I used for this is utilizing the ChatGPT tool, as shown in the section.

3.3 ChatGPT

ChatGPT [15] is an AI language model, with the purpose of generating human-like text based on the prompts and questions provided. This tool is free to use, with an optional premium subscription costing 20$ per month with perks I show in another point down the line.

This tool was developed by OpenAI, a research organization focused on artificial intelligence and machine learning advancements. Their goal is to help advance humankind through the development and deployment of safe and beneficial cutting-edge artificial intelligence technologies.

The task of being a programmer requires coding with built in functions from packages. There is an immense pool of functions in code, each doing a particular task, and each with a fixed name. For a programmer to do their profession, they need to find ways to implement the features at hand through the use of these functions. The issue is that we are human, and can not remember every function that we have encountered, nor can we know functions we have never seen before. To solve this issue, we search on web browsers the methods we are intending to use, to check the documentation on how to operate them, check on guides, or learn new alternatives.

During the development of Math-Masters, I utilized ChatGPT on a regular basis to quickly answer the small programming questions I had, as shown in the figure 3.3.

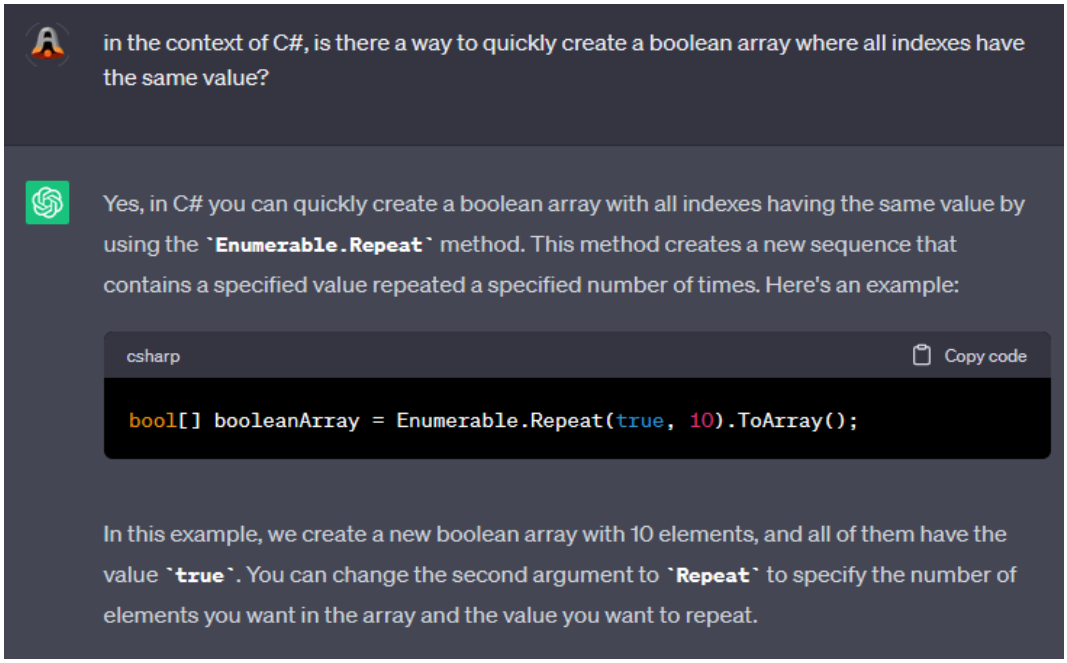


Figure 3.3: Screenshot of the website showing the answer of a question I asked

These answers about programming are usually structured as the following: The first paragraph is the actual answer, stating the name of the method that will perform as asked. Second, show an example of said method inside a code block (sometimes the text formatting of this code block will incorrectly guess what type of programming language is being shown, though this formatting mistake does not influence the answer made by the bot, as shown in the figure 3.4). Third, explain the parameters or interactions between methods. With these three steps combined, the result is an answer written in language that humans can understand that shows what to apply and how to apply. The figure 3.3 referred before reflects this structure.

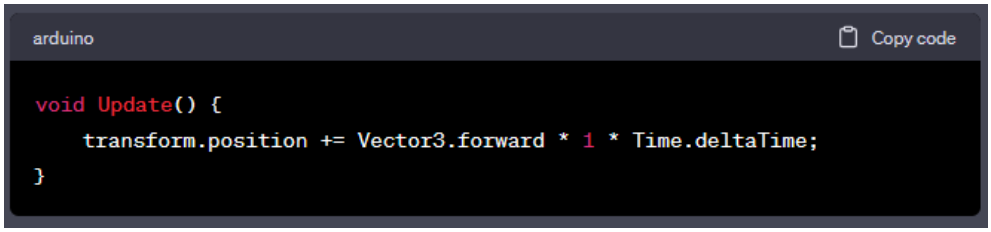
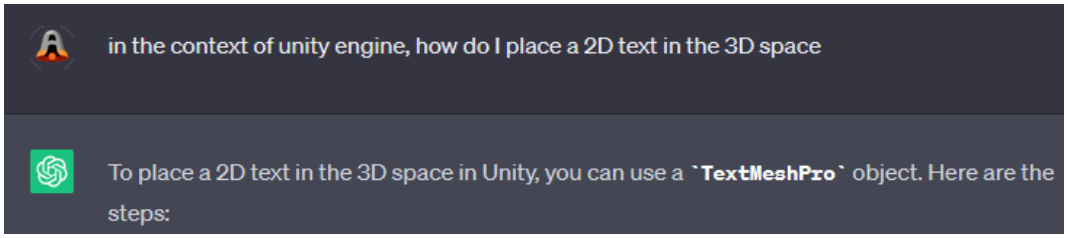
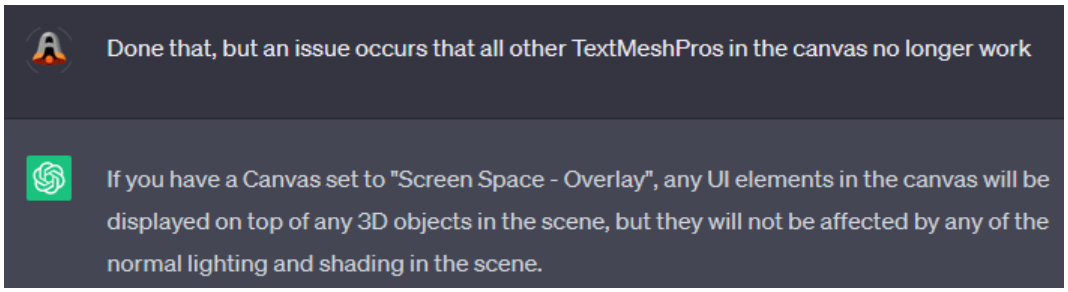


Figure 3.4: Screenshot of the website displaying Arduino instead of C# in a code block

Another feature that makes ChatGPT stand out is remembering what the user stated and the information the bot itself showed. This streamlines the back-and-forth between user and bot, and conveniently simulates a conversation, as shown in the figure 3.5.



(a) Starting with a question with context



(b) Prompting another question afterwards without context

Figure 3.5: Screenshots of the website showing the capability to hold conversations

ChatGPT writes these answers in under 10 seconds (scaling in proportion to the size of the answer). This speed is relevant as it shortens the amount of time spent searching for answers that can instead be used productively in the programming workflow. Due to its nature as a free-to-use tool, ChatGPT can occasionally throttle in responsiveness when use is in high demand, even cutting access during peek use. In these situations, the user can opt for the premium service that grants availability during high demand and faster response speed than the free version.

Nevertheless, ChatGPT does not give accurate answers at all times. As a matter of fact, it will repeatedly remind the user that the information they have only dates up to September 2021, and some answers can get biased according to the patterns in its algorithm. Users need to be aware of this degree of uncertainty, and have to check for validity every now and then or tolerate the failed attempts at understanding the question. ChatGPT will repeat the same mistake on a chain of answers on numerous occasions.

For these reasons, I conclude that this is a beneficial tool for programmers, despite being seldom faulty for the current version. As time progresses, better versions will be released and the training data will be updated closer and closer to the present. I see myself using this tool in my professional career as a programmer.

Chapter 4

Design and Development of Math-Masters

4.1 Game Concept

Math-Masters is a point-and-click deck-building and partially rogue-lite third-person educational game with the purpose of teaching the multiplication operation in mathematics. It is themed around medieval fantasy, possessing monsters on one hand and magic spells on the other. The first area players will encounter is the central hub, called Samos Town. This location is a safe zone, meant for the player to explore and have access to shops to change appearance or upgrade their stats. The exit gate from this town will take the player to the several levels to face the monsters and gain gold as reward. Player attacks against said monsters are represented by cards.

The educational aspect of this game was created according to the methodology from [4]. In order to gain rewards and progress through the game, the player must put on the effort of solving the mathematical questions that appear on screen. In order to answer, the play must click on one of the cards that appear in their hand of cards at the bottom side of the screen. Answering correctly will allow the player to strike, answering incorrectly will be met with retaliation from the monster at the opposite side of the screen. Each attempt will reshuffle the hand of cards.

The player may also resort to the spell cards that act as assistance this game offers.

These cards reduce the difficulty of the game but at the same time are a small layer of complexity. For balancing purposes, these spells are restricted in use, meaning the player has to make decisions on when is the best time to utilize them. Burning through these cards during the start of a level may prove as a mistake since the difficulty increases as the player progresses through the level. In addition, these spell cards are not guaranteed to be drawn into the hand.

The difficulty increase is in the multiplication questions getting progressively harder and the monsters having a bigger health pool. Proportionally, defeating harder difficulty monsters yields greater amounts of in-game currency.

The rogue-lite aspect of this game is that progression is only saved upon clearing a level, meaning that defeating monsters and then restarting or quitting to Samos Town will reset the amount of currency owned back to the amount the player had before starting the level.

4.2 Game Requirements and Use-Case

This game requires internet connection for the entire duration of the game session, so as to be connected with the external database from PlayFab. Currently there is only a build for the Windows Operating System, working in both Windows 10 and Windows 11. In order to Login, the player must have a way to input text onto the credential fields. Due to the nature of point-and-click genre, this game requires a mouse or any alternative to move the mouse pointer.

The figure 4.1 shows this game's Use-Case.

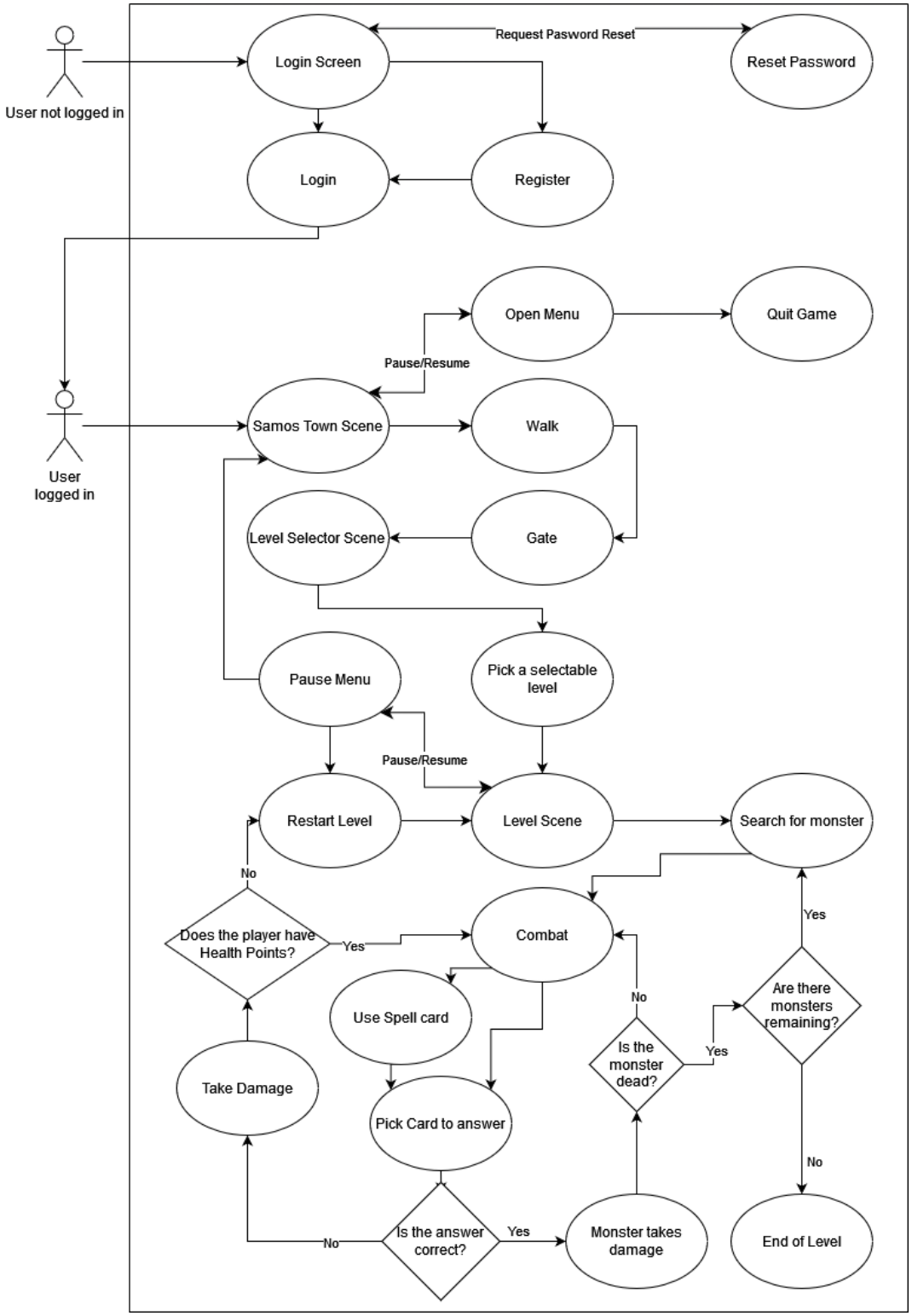


Figure 4.1: User Case Diagram of Math-Masters

4.3 Scenes and their respective mechanics

Math-Masters contains 9 scenes, and the build order is as following:

- 0. Login Screen
- 1. Samos Town
- 2. Level Selector
- 3 - 7. Level 1 to 5
- 8. Ending Screen

4.3.1 Login Screen

Opening the game through the executor will first open the login screen, as shown in the figure 4.2, this scene functions mainly through the PlayFabManager script.

Figure 4.2: Screenshot of the login screen

Since this game requires a connection to its database, it requires the internet in order to function and keep track of the players' progress.

Prompting the player to register by typing credentials into the name text field, the email text field and the password text field, each of these fields are a variant of the TMP package, the TextMeshPro - Input Field [16]. Clicking on the register button will grab

the information in the text fields and create a **RegisterPlayFabUserRequest** type of object. Through this object, the information is then sent as input to PlayFab's database by utilizing the **PlayFabClientAPI.RegisterPlayFabUser** method. On success, the user account is created and the player is logged in additionally. On failure, will show the error on the debug message, more on that further below.

If the user already has an account, they can type the corresponding email and password credentials. On clicking the login button, the information is sent to a **LoginWithEmailAddressRequest** which is used in the **PlayFabClientAPI.LoginWithEmailAddress** method, as shown in the figure 4.3. The other option would be to use LoginWithPlayFab with username instead of email as part of the required credentials to login, the decision is up for the developers' preference as either work the same way. PlayFab has many alternatives to logging through external accounts, such as integration with Apple, Google, Facebook and Steam accounts. For the purposes of this project, users can only login through accounts made in PlayFab.

```
public void LoginButton()
{
    var request = new LoginWithEmailAddressRequest
    {
        Email = emailInput.text,
        Password = passwordInput.text
    };
    PlayFabClientAPI.LoginWithEmailAddress(request, OnLoginSuccess, OnError);
}
```

Figure 4.3: Sample of the **PlayFabManager** script code

The last button of the login screen is the reset password button, which only requires the email text field. By utilizing the **PlayFabClientAPI.SendAccountRecoveryEmail** method, it will send a mail to the received email, allowing the user to change the password by opening an encrypted page on PlayFab's website, asking to type a new password and repeat.

At the bottom of the login screen, there is a debugging text displaying a message according to the input of the user. It will display either success or failure to connect, any mistake around the information used in the text fields, successful register, successful login and reset email sent.

An added feature to this screen is the use of "TAB" and "SHIFT" to switch between text fields and buttons. TAB goes down on the screen while SHIFT goes up. This is enabled through the ChangeInput script. This was an experiment in using the EventSystem to have an alternative way to cycle through the various HUD elements on the canvas.

After a successful login or registration and before switching scenes, the **PlayFab-Manager** script will grab from the user's account their username, "MaxLevelComplete" value and "Money" value stored in the Player Data, referred to as Titles in PlayFab terminology. Titles: where game-specific data associated with individual players is stored. It can be directly accessed in the PlayFab website (as shown in the figure 4.4), allowing developers to manage player profiles, track progression, enable personalization, and derive valuable insights to enhance the overall gaming experience.

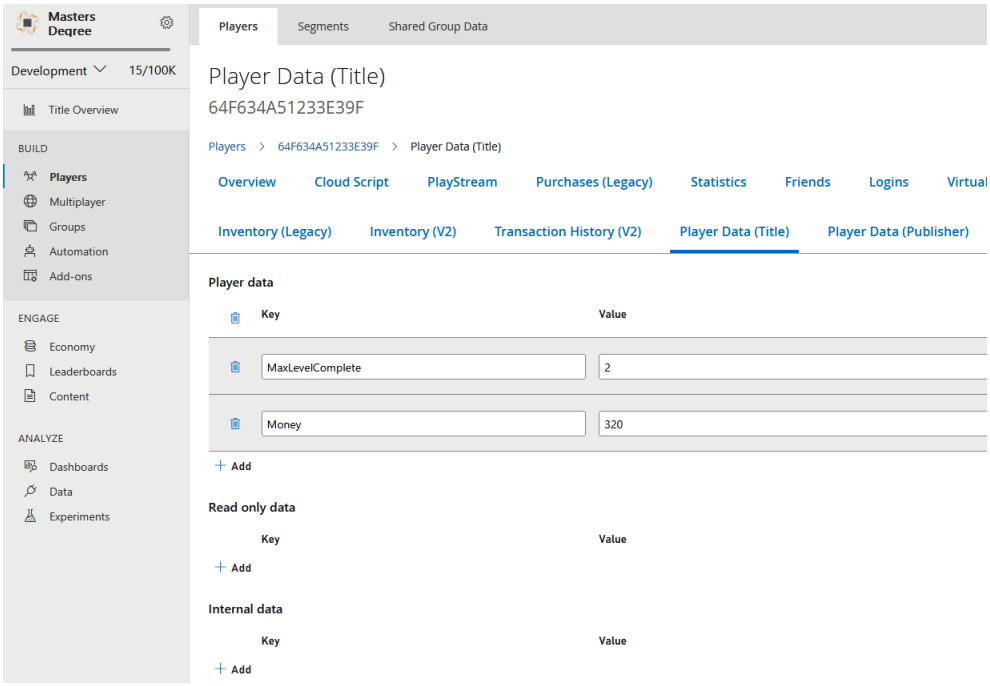


Figure 4.4: Screenshot of PlayFab's website

After successfully parsing the fetched values, they are stored in the only Static class of this project - **Static_PlayerProfile** script. The code from the class **Static_PlayerProfile** script is shown in figure 4.5

```
public static class StaticPlayerProfile
{
    5 references
    public static string PlayerName { get; set; } = "No player name";
    6 references
    public static int MaxLevelComplete { get; set; } = 0;
    6 references
    public static int Money { get; set; } = 0;

    4 references
    public static int MoneyToAdd { get; set; } = 0;
}
```

Figure 4.5: Screenshot of **Static_PlayerProfile** script code

The reason for storing these important values in a static class is that this class will not be unloaded with scene transitions, therefore the variables are available to get and set at any point during runtime.

4.3.2 Samos Town

The scene after login is Samos Town, a safe Hub with no enemies where the player can explore the location, as shown in the figure 4.6.

Figure 4.6: Screenshot of the Samos Town Scene

Opening the scene will encounter a tutorial stating what a player can do as shown in the figure 4.7. Clicking on the "Next" button at the bottom of that window will clear

it away.

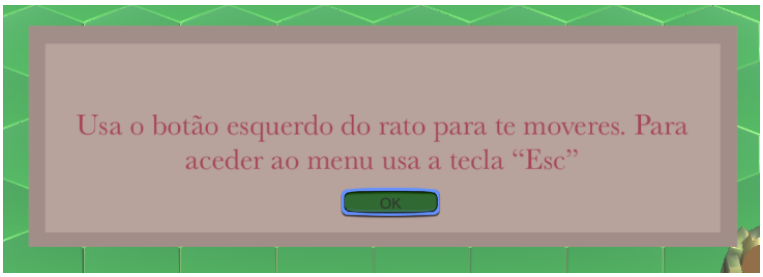


Figure 4.7: Screenshot of the Samos Town Scene

This scene contains two types of movement, either by using the WASD keys or clicking with the mouse on the terrain of the area. Each type uses different methods of locomotion in the code. The player gameObject possesses two different components to allow these two different methods of movement: a RigidBody and a NavMeshAgent.

- Point-and-click movement happens when the player clicks on terrain eligible to be walked on. A Raycast is made to detect the terrain, upon success will use the coordinates of the terrain where the player clicked to make destination for the NavMeshAgent and allow it to reach to the destination according to the computer generated path. For this to be possible, the walkable ground had to be baked with a navigation area. The figure 4.8 shows the part of the code that detects mouse input.

```
// Mouse Click Detection (Only happens if WASD is not being pressed)
else if (Input.GetMouseButtonDown(0))
{
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

    if (Physics.Raycast(ray, out ClickedLocation, Mathf.Infinity))
    {
        // Clicked on Terrain
        if (ClickedLocation.collider.CompareTag(groundTag))
        {
            NavigationAgent.SetDestination(ClickedLocation.point);
            isMovingToClickedLocation = true;

            Instantiate(GO_DisplayClickedLocation, ClickedLocation.point, Quaternion.identity);
            FootstepsSound(true);
            MainScript.AnimationScript.ToggleAnimation("isWalkingFWD", true);
        }
    }
}
```

Figure 4.8: Sample of the **PlayerMovement** script code

In addition, where the mouse clicks, a visual indicator is instantiated. This indi-

cator is a prefab gameObject with the **GO_ClickedLocation** script that destroys itself after its animation is over. The figure 4.9 shows this indicator.

Figure 4.9: Screenshot of gameplay showing the visual indicator

- WASD Movement happens when the player presses and/or holds the one of the namesake keys. Pressing "W" will make the character walk north, "S" will make the character walk south, "A" will make the character move west and "D" will make the character move East (this is the conventional movement input system for videogames). Pressing these keys will rotate the player character by changing the direction the gameObject is facing (modifying the Vector3 of the transform.forward). This movement will shut off any navigation paths, so as to force the player gameObject move according to the input from the player. Not stopping the navigation agent results in a clash between the two movement methods. Manual movement is achieved by modifying the velocity of the RigidBody. The following figure 4.10 shows this part of the code.

```
// WASD Movement
if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.D))
{
    // Stop any mouse movement
    isMovingToClickedLocation = false;
    NavigationAgent.velocity = Vector3.zero;
    NavigationAgent.isStopped = true;
    NavigationAgent.ResetPath();

    Vector3 rightMovement = right * movementSpeed * Time.deltaTime * Input.GetAxis("Horizontal");
    Vector3 upMovement = Playerforward * movementSpeed * Time.deltaTime * Input.GetAxis("Vertical");
    Vector3 heading = Vector3.Normalize(rightMovement + upMovement);

    if (heading != Vector3.zero) transform.forward = heading;

    rb.velocity = heading * movementSpeed;
```

Figure 4.10: Sample of the **PlayerMovement** script code

Samos Town Scene has a pause menu upon tapping the "ESC" key. While the menu is open, the player is unable to interact with the game outside the pause menu, and the

player gameObject stops moving, regardless of what movement type was acting on it. This pause menu has the resume button, quit game button, an "About Me" button and a "Customize character" button. Both the "About me" and "Customize character" buttons do nothing upon clicking because the features behind them were not implemented.

The player also has the ability to zoom in and out with the third person camera by scrolling with the mouse wheel. This is controlled by the **PlayerCamera** script, which its main purpose is to increase or decrease the orthographicSize of the main Camera according to the input from the player. This is one of the few methods continuously being "listened" by the main Script in the centralized Update method. More on that at a later point.

To change to a level, the player has to move to the southmost part of Samos Town. There is a open gate there, with an arrow moving up and down to signal the player where to proceed. When the player gameObject moves to the gate, it collides with an invisible wall the size of the gate that triggers the scene transition to the Level Selector.

With further development, more features would be added to Samos Town to give it purpose: a window to change the appearance of the character, a settings window to allow the player to change sound volume, graphics and more. As it stands, it is a demonstration on what the location and movement would be like had the game taken different routes of design.

4.3.3 Level Selector

Players will be able to choose the unlocked levels. A player that started playing the game for the first time will only be able to play the first level.

Unlocked levels are marked as blue while the locked levels are gray, as seen in the figure 4.11.

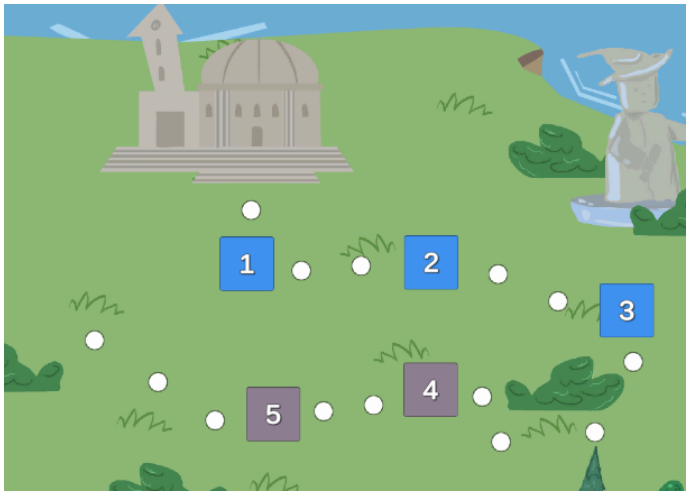


Figure 4.11: Screenshot of the Level Selector Scene

This scene's functionality comes from the buttons representing the levels. Each button has the LevelSelector script unique to this scene, where the public variable "level" changes. The code from the class LevelSelector script is shown in the figure 4.12

```
public class LevelSelector : MonoBehaviour
{
    2 references
    public int level;

    0 references | Unity Message
    private void Awake()
    {
        if (level > StaticPlayerProfile.MaxLevelComplete + 1)
        {
            transform.GetComponent<Button>().interactable = false;
        }
    }

    0 references
    public void OpenScene()
    {
        SceneManager.LoadScene("Level" +level.ToString());
    }
}
```

Figure 4.12: Sample of the **LevelSelector** script code

On Awake, each of them will check if the player is allowed to select the level attached to the button, by turning themselves non interactable if that is not the case.

When not interactable, the gameobject will change its visual appearance to the color defined in "Disabled Color", as shown in the figure 4.13.

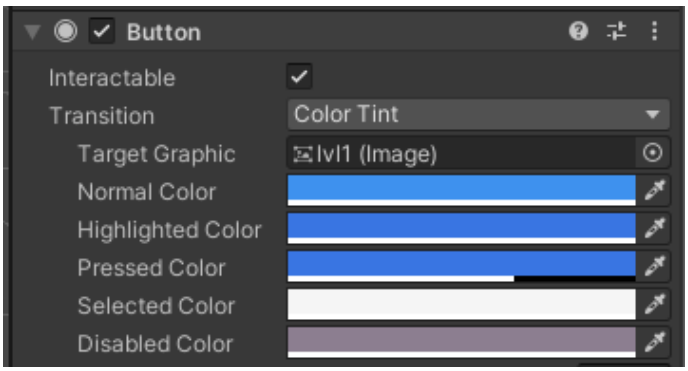


Figure 4.13: Screenshot of the Button gameObject component

This step not only disables the user to press the button, but also changes appearance. These buttons, on click, will call the OpenScene method, changing the scene to the one they represent.

4.3.4 Levels

Levels are where the player encounters monsters (as shown in the figure 4.14), enters combat, defeats them, gains in game currency, and finishes the level.

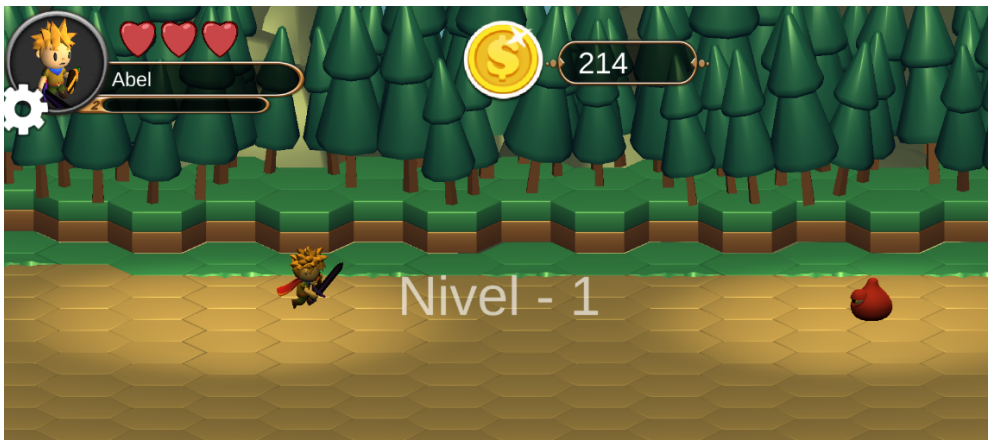


Figure 4.14: Screenshot of the gameplay level before encountering an enemy

The bulk of the coding surrounds these scenes as it is here where the most important mechanics lie. The more relevant scripts that make this happen are the **PlayerCombat**,

PlayerDeck and **PlayerQuestion**. The way the coding works in the levels as a whole is as a loop, and is described as the following:

1. Approach enemy
2. New turn
3. Pick a card
4. End of Level

4.3.4.1 Approach enemy

As the level starts, a crucial part that happens on Awake is the registration of the enemy group in the Scene Hierarchy, placing them in a "queue" List. Each enemy has variables describing their difficulty and amount of in-game currency to grant upon being defeated.

The player gameObject moves to towards the first enemy in the queue. The player has no input over the player gameObject (in other words, no control over the movement), as these movements are utilizing NavMeshAgent as shown in the figure 4.15.

```
public void ForceMoveToLocation(Vector3 WhereToGo)
{

    NavigationAgent.ResetPath();
    NavigationAgent.SetDestination(WhereToGo);
    isMovingToClickedLocation = true;

    FootstepsSound(true);
    MainScript.AnimationScript.ToggleAnimation("isWalkingFWD", true);
}
```

Figure 4.15: Sample of the **PlayerMovement** code

When the player, through the NavMeshAgent, reaches a certain distance from the enemy, it will stop the movement. It is at this point that the combat begins and the first turn begins, as shown in the figure 4.16. When the player encounters their first enemy, a brief tutorial appears just like in Samos Town.

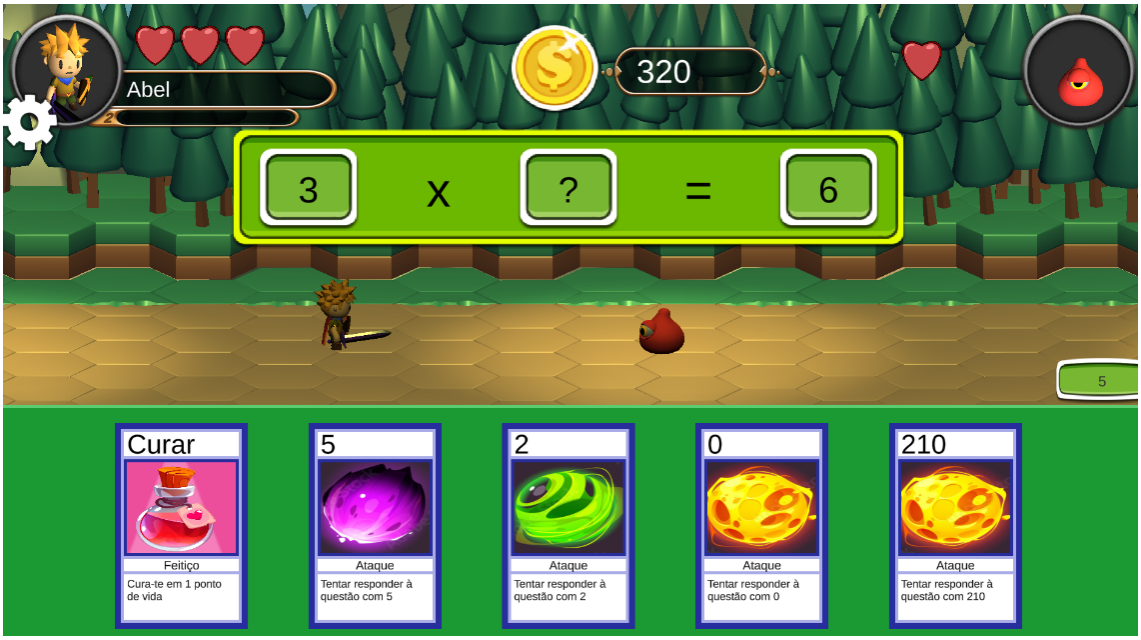


Figure 4.16: Screenshot of the gameplay combat

4.3.4.2 New turn

The start of each turn begins by generating a mathematical question according to the premade selection of questions in the game's design files (more on that in the section *Interconnection between Design and Development*). This process is done through the **Generate_NewQuestion** method, which provided the current enemy encountered, will randomly select the question according to said enemy's difficulty level. Afterwards, will randomly pick a field to leave empty to become the missing element that the player must fill in order to progress. To finish, it sends the information to the HUD elements displaying the question, and returns the answer of the empty field to be used later in the turn. The figure 4.17 shows the method.

```
public int Generate_NewQuestion(Enemy CurrentEnemy)
{
    int answerToQuestion = 0;
    int pickATier = Random.Range(CurrentEnemy.MinimumTier, CurrentEnemy.MaximumTier + 1);
    int randomInt = Random.Range(0, Dic_QuestionsByTier[pickATier].Count);
    QuestionPanel.Question question = Dic_QuestionsByTier[pickATier][randomInt];

    int calculatedResult = Calculate_Result(question);

    string FieldToEmpty = "N/A";
    randomInt = Random.Range(0, 3);
    switch(randomInt)
    {
        case(0):
            FieldToEmpty = "FirstNumber";
            answerToQuestion = question.FirstNumber;
            break;
        case(1):
            FieldToEmpty = "SecondNumber";
            answerToQuestion = question.SecondNumber;
            break;
        case(2):
            FieldToEmpty = "Result";
            answerToQuestion = calculatedResult;
            break;
    }

    QuestionScript.Create_Question(question, FieldToEmpty, calculatedResult);
    return answerToQuestion;
}
```

Figure 4.17: Sample of the **PlayerQuestion** code

The green strip filling the bottom half of the screen is where the hand of cards the player will be. After generating the question, the player draws a new hand of cards. This process is done by the **Generate_NewHand** method, that, provided the answer to the question, will first locate where to place the cards on screen, second pick one spot in the hand of cards to place the answer card of the question, third fill the remaining spots with random cards without causing duplicates. The figure 4.18 shows this method's code. While the cards are being drawn (coded animation of appearing one by one from left to right), the players is unable to interact with the combat.

```
public void Generate_NewHand(int answerToQuestion)
{
    Vector3[] cardCoordinates = Generate_CardCoordinates();
    int randomInt = Random.Range(0, cardCoordinates.GetLength(0));

    MainScript.Bool_InterruptableCoroutineIsHappening = true;
    MainScript.InterruptableCoroutine = StartCoroutine(Coroutine_Generate_NewHand(answerToQuestion, cardCoordinates, randomInt));
}
```

Figure 4.18: Sample of the **PlayerDeck** code

4.3.4.3 Pick a card

There are two types of cards drawn: AttackCard and SpellCard. Both of these card types are classes under the BaseCard class of the **BattleCard** script. Unifying these classes under the same banner simplifies storing the objects.

With the cards drawn, the player can now decide which card to pick. The cards that appear on screen have a button component, that when clicked will call the method **ClickedCard**, as shown in the figure 4.19.

```
public void ClickedCard()
{
    if(!MainScript.Bool_InterruptableCoroutineIsHappening)
    {
        if(AttackInfo != null)
        {
            MainScript.CombatScript.EndTurn(AttackInfo);
            GameObject.Destroy(gameObject);
        }
        else if(SpellInfo != null)
        {
            MainScript.CombatScript.SpellCast(SpellInfo);
            //gameObject.SetActive(false);
            GameObject.Destroy(gameObject);
        }
```

Figure 4.19: Sample of the **BattleCard** code

If the card clicked on is an AttackCard, then the turn progresses and the consequences happen according to the answer. Alternatively, the player can click on a SpellCard before picking an AttackCard, assuming one was drawn. SpellCards will influence the outcome of the consequences, as shown in the figure 4.20. Spellcards are restricted by an amount, reaching this limit will remove functionality of the SpellCards.

```
public void SpellCast(BattleCard.SpellCard spell)
{
    if(AmountOfSpellsThePlayerCanUse <= 0) return;

    if(!ASpellWasCast)
    {
        switch(spell.SpellType){
            case("Block"):
                    //"Neste turno bloqueias o ataque se errares";
                    BlockAttack = true;
                break;
            case("Heal"):
                    //"Cura-te em 1 ponto de vida";
                    MainScript.HealthScript.Heal(1);
                break;
            case("DoubleAttack"):
                    //"Durante este turno, causa o dobro do dano se acertares";
                    PlayerAttacksTwice = true;
                break;
        }
        ASpellWasCast = true;
        AmountOfSpellsThePlayerCanUse--;
        spellCounter.UpdateSpellCounter(AmountOfSpellsThePlayerCanUse);
    }
}
```

Figure 4.20: Sample of the **PlayerCombat** code

After selecting an AttackCard, the value from that card is compared to the answer of the question saved. If it is the same, then the player successfully deals a blow onto the enemy, reducing their health by 1. If the DoubleAttack SpellCard was picked, then the player deals 2 damage instead. Incorrectly guessing the answer will make the enemy strike at the player, lowering their health by 1 unless a Block SpellCard was picked. If the player's health is dropped to 0, then a "defeat" menu appears, which is identical to the pause menu except the player is forced to either leave or restart the level. The user can prepare for this event by selecting a Heal SpellCard to increase their current health.

Before proceeding, the game will reset the SpellCard picked and discard the current hand of cards by calling the **Discard_Hand** function from the **PlayerDeck** script.

The figure 4.21 shows the function that manages the end of turn.

```
public void EndTurn(BattleCard.AttackCard attack)
{

    bool isCorrect = MainScript.QuestionScript.AttemptAtAnswer(attack);

    if(!isCorrect)
    {
        CurrentEnemy.EnemyAttack();

        if(!BlockAttack && MainScript.HealthScript.TakeDamage())
        {
            MainScript.PauseMenuScript.Pause(true);
            return;
        }
    }
    else
    {
        CurrentEnemy.TakeDamage(PlayerAttacksTwice);
    }

    ASpellWasCast = false;
    BlockAttack = false;
    PlayerAttacksTwice = false;
    MainScript.DeckScript.Discard_Hand();
}
```

Figure 4.21: Sample of the **PlayerCombat** code

After the enemy is either being attacked or attacking, when the animation ends its **Enemy** script will communicate with **PlayerCombat** script, by ordering a new turn to start. However, if the enemy is killed, it will instead say that the fight is over and the player gameObject should go after the next enemy. The figure 4.22 shows the described function, this function is called whenever an animation is completed by the enemy, excluding idle animation. This is where the combat loop repeats, by starting a new turn and prompting the player with a question and drawing a new hand of cards, only stopping when either the player or the enemy is defeated.

```
public void AnimationEnded_ContinueTheGame(int option)
{
    switch(option)
    {
        case(1): // Hit
            if(AboutToDie)
            {
                return;
            }
            MainScript.CombatScript.NewTurn();
            break;
        case(2): // Die
            OnDeath();
            MainScript.CombatScript.NextEnemyFight();
            break;
        case(3): // Attack
            MainScript.CombatScript.NewTurn();
            break;
    }
}
```

Figure 4.22: Sample of the **Enemy** code

The way I implemented the continuation of the mechanics at the end of the enemy animations is by adding events to the frames of the enemy animations, more specifically at the last frame, as shown in the figure 4.23. These events will look for public methods from the scripts attached to the gameObject where the animation is taking effect. Naming the function shown in the previous figure 4.22 and giving it the required argument in the right field will suffice for the intended.

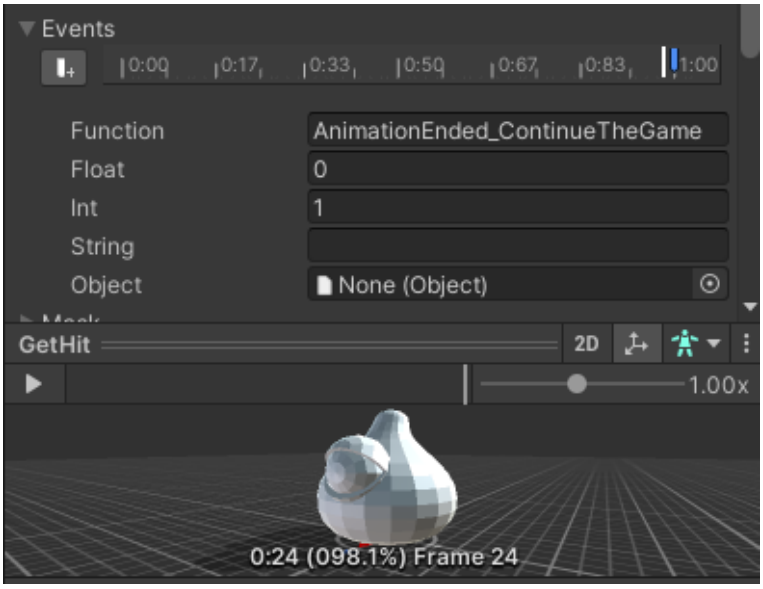


Figure 4.23: Enemy's Hit animation in the Inspector

After the enemy is defeated, the player searches for the next enemy in the queue, causing the level loop of encountering an enemy, defeating it, and encountering the next enemy. Whenever an enemy is defeated, the index of the next enemy to fight is incremented to update the current enemy for an undefeated one. When this index is above the amount of enemies there are in the level, means there are no more enemies to fight against, granting a level completion. The figure 4.24 shows the function that manages this.

```
public void NextEnemyFight()
{
    EnemyIndexInList++;

    if (EnemyIndexInList < ListOfEnemies.Count)
    {
        SetUpCombat();
        SetUpEnemyProfile();
    }
    else LevelComplete();
}
```

Figure 4.24: Sample of the **PlayerCombat** code

4.3.4.4 End of Level

On level completion, a completion menu fills the screen. The Player Data (as shown in the figure 4.25) "Money" is updated with the result of the sum between the current amount of money saved and the amount of money gained by defeating the monsters of the completed level. This quirk was made to prevent the exploit of reloading the level and keeping the money without completing the level. If the finished level was a first time completion, the Title "MaxLevelComplete" is incremented, so as to allow the player to select the following level if they decide not to continue immediately, and select the new level through the Level Selector at a later time.

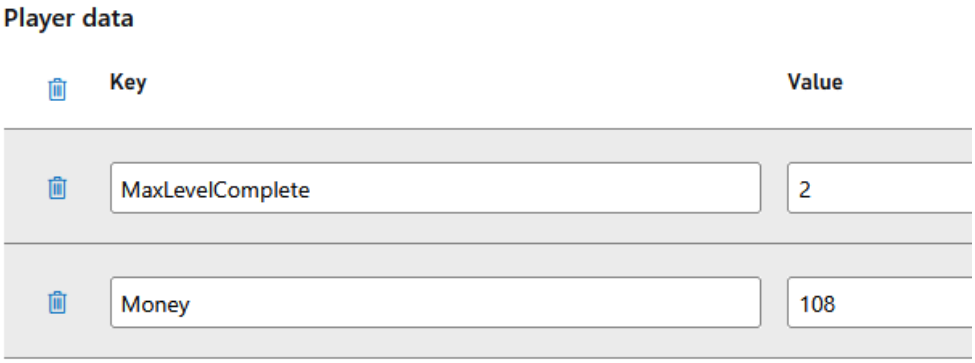


Figure 4.25: Player Data in the PlayerFab website

The process to update Titles requires the creation of an **UpdateUserDataRequest** object, which contains a dictionary with the names of the titles to update and their corresponding new values. Afterwards, this object is used in the **PlayFabClientAPI.UpdateUserData** method, for sending the information to the database. The figure 4.26 shows this function.

```
public void SetPrimaryTitleData()
{
    var request = new UpdateUserDataRequest
    {
        Data = new Dictionary<string, string>
        {
            {"MaxLevelComplete", "" + StaticPlayerProfile.MaxLevelComplete},
            {"Money", "" + StaticPlayerProfile.Money}
        }
    };
    PlayFabClientAPI.UpdateUserData(request, OnSetPrimaryTitleDataSuccess, OnError);
}
```

Figure 4.26: Sample of the **PlayerMainScript** code

Login Screen is the only scene that fetches information from the database. For the remainder of the scenes, each time the information stored in the *Static_PlayerProfile* is changed, the changes are sent to the database. This results in not needing to fetch any information from the database since the one stored by the client side is updated.

During this update of the database, the player is unable to give any input to the game because of the completion menu so as to not interrupt the update. A message will display the current status of the update, allowing the user to retry the update of the information in case it didn't work, or letting the user leave. On success, allows the player to return to Samos Town or to proceed to the next level, as shown in the figure 4.27.

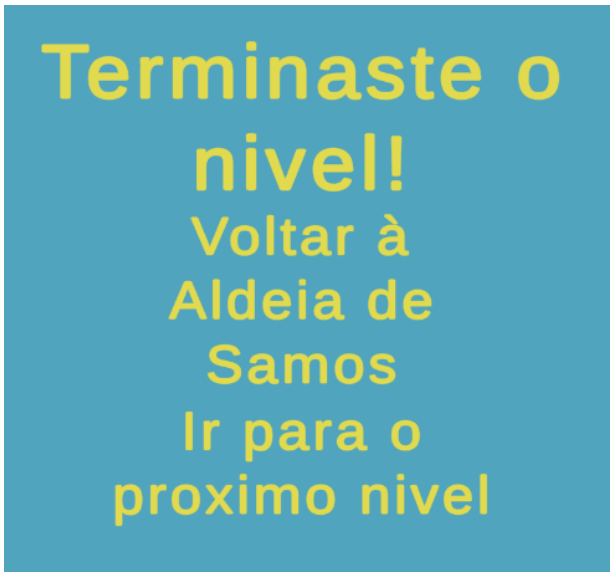


Figure 4.27: Screenshot of the gameplay completion screen

The player can open the pause menu utilizing the "ESC" key at any point of the level excluding the defeated screen or the completion screen. This will pause the **Time.timeScale** to zero (pausing any animation, NavMeshAgent movement and coroutine) and turning a Boolean "GameIsPaused" to true which will disable the centralized Update Method, more on that at a later point.

4.3.5 End Screen

End screen is composed of a menu in similar line to the one in the levels, with the exception that it only has a single button to return back to Samos Town, as shown in the figure 4.28.

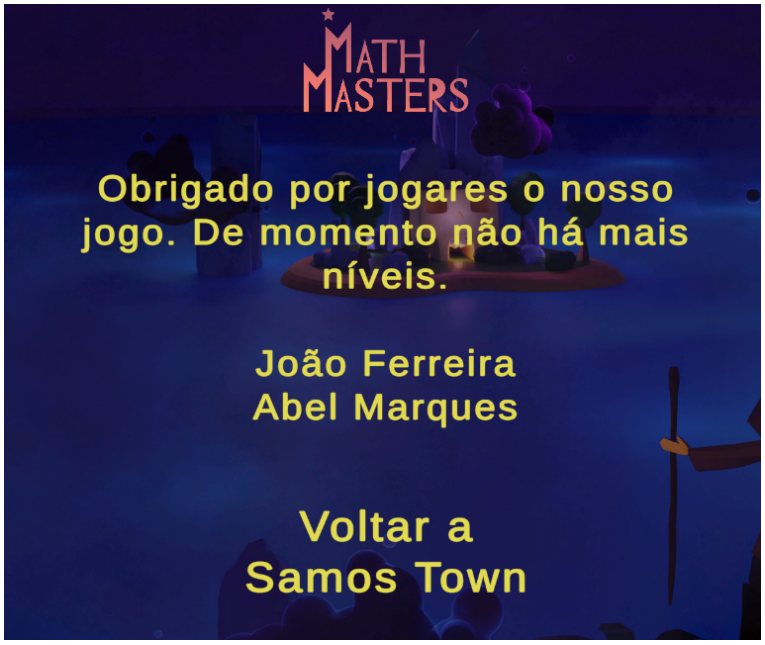


Figure 4.28: Screenshot of the gameplay endscreen

4.4 Interconnection between Design and Development

4.4.1 Data from the main mechanics

Math-Masters contains 3 separate types of object data that shape the gameplay: AttackCards, SpellCards and Questions. All three types of data objects are saved like objects in three separate files, one for each type, containing all variables and their information, as shown in the figure 4.29.

Figure 4.29: Sample of the **SpellCards** JSON file

From this arises an issue in that the built in deserializer of Unity - JsonUtility - only supports a limited set of data types such as arrays, dictionaries and simple types like integers and strings. In order to handle more complex JSON data, I had to resort to a new parsing library called "Newtonsoft.Json". This new library provides me with the **JsonConvert.DeserializeObject** method, which directly deserializes information from the JSONs into the objects. The figure 4.30 shows that the code provides the deserializer with a text file in JSON format, and make it return an array of objects.

```
/*
    ### Attack Cards
*/

TextAsset textAsset = Resources.Load<TextAsset>("Attackcards");
Array_AttackCards = JsonConvert.DeserializeObject<BattleCard.AttackCard[]>(textAsset.text);
```

Figure 4.30: Sample of the **PlayerDeck** code

As described in the "Scenes and their respective mechanics" section, what the code does when a turn begins is check the information from the enemy to decide. Each question has an unique tier, and each enemy has a minimum tier and a maximum tier Public variable. The code will randomly pick a question between the minimum and maximum tiers of the enemy, the boundaries are inclusive. The figure 4.31 shows an example of information from an enemy.

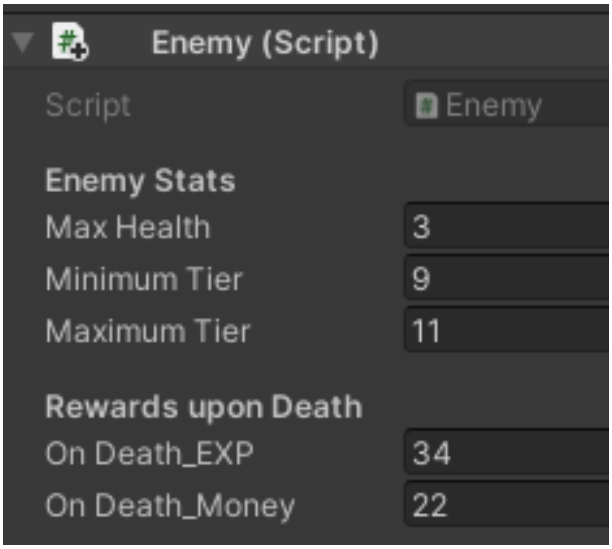


Figure 4.31: Screenshot of the Enemy's script values in the Inspector

The use of JSON files to store data gives the liberty to the game designer to adjust/change the game contents without the need to adjust/change the code of the game. With a twist in that SpellCards need to have coded lines in the **PlayerCombat** script as shown in the previous figures 4.20 and 4.19.

4.4.2 The levels

The way the game is built, each level acts almost independently from any other scene, excluding the login screen which fetches the Player Data from the database. This results in the ability to add or remove levels without impacting the rest of the game's code. The only step required is to add, remove or update level buttons in the Level Selector (as shown in the previous figure 4.11) accordingly. The End Screen scene is the limit index of the build order to redirect the user back to Samos Town, regardless of what prior level was completed. The figure 4.32 shows this modularity.

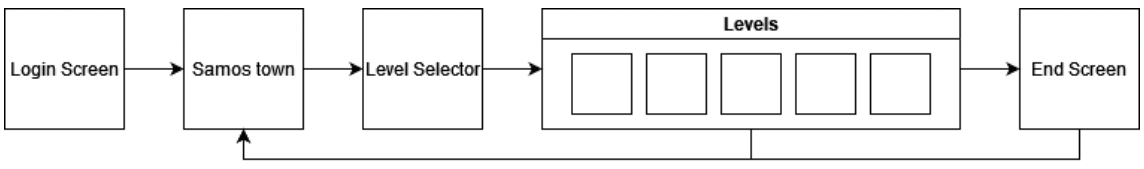


Figure 4.32: Diagram of the Scene modularity

There is an implemented tool embedded in the **PlayerDeck** script that allows the

Designer to change the dimensions of the cards and how many cards are going to appear on screen, in real time outside the game. In addition, it was through this debugging method that allowed a faster implementation of the hand of cards. This tool is called DrawGizmos, as seen in the figure 4.33 and the elements in the Inspector shown in the figure 4.34.

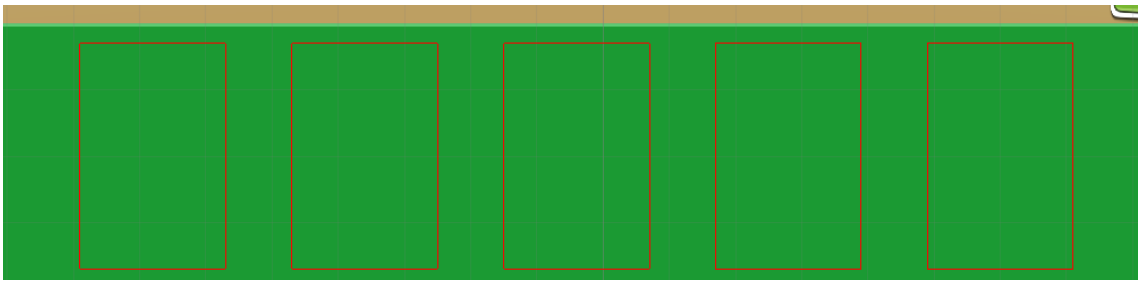


Figure 4.33: Canvas with DrawGizmos toggled on

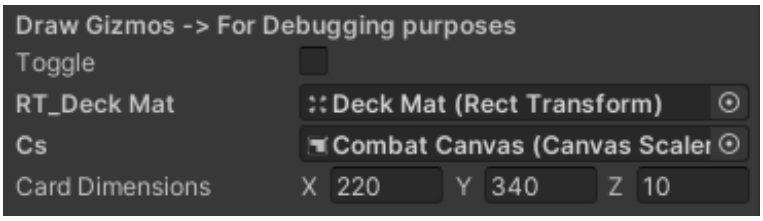


Figure 4.34: PlayerDeck component in the Inspector

With all said from this section, this methodology of allowing modification to the game's content without changing the code and making it modular gives the Designer autonomy. They are able to change the design of the game and expand it without the presence of the Programmer.

4.5 Hierarchy of the player scripts

Each implemented feature regarding the player has its own script. Those scripts inherit from the parent class **Parent_PlayerScript**, and they are all attached to the player gameobject.

On scene load, the main script will run the **Run_At_start** method of each player script, as shown in the figure 4.35. Since Samos Town is meant to not have any combat

mechanics, the **PlayerCombat**, **PlayerDeck** and **PlayerQuestion** are left aside. Scenes without player scripts do not need to be checked.

```
#region Script Initializing
    MovementScript = GetComponent<PlayerMovement>();
    HealthScript = GetComponent<PlayerHealth>();
    AnimationScript = GetComponent<PlayerAnimation>();
    CameraScript = GetComponent<PlayerCamera>();
    CombatScript = GetComponent<PlayerCombat>();
    MoneyScript = GetComponent<PlayerMoney>();
    DeckScript = GetComponent<PlayerDeck>();
    QuestionScript = GetComponent<PlayerQuestion>();

    MovementScript.Run_At_Start();
    HealthScript.Run_At_Start();
    AnimationScript.Run_At_Start();
    CameraScript.Run_At_Start();
    if(int_CurrentScene != 1) CombatScript.Run_At_Start();
    MoneyScript.Run_At_Start();
    if(int_CurrentScene != 1) DeckScript.Run_At_Start();
    if(int_CurrentScene != 1) QuestionScript.Run_At_Start();
#endregion
```

Figure 4.35: Sample of the **PlayerMainScript** code

Due to the controlled nature of turn based interactions implemented in our core gameplay, I opted for a single centralized Update method in the **PlayerMainScript**, for all the Player scripts. This script does not have heritage, however all other player scripts have a dedicated variable to store the pointer to the main script. This centralized Update method picks between manual or automatic movement according to the scene, as well as updating the main camera's position relative to the player gameobject. These interactions are only allowed when the game is not paused, as shown in the figure 4.36.

```
private void Update()
{
    if(!PauseMenuScript.GameIsPaused)
    {
        // ## Only usable in Samos Town (Scene 1); WASD/Mouse-Click Movement and character direction Component.
        if(int_CurrentScene == 1) MovementScript.Move();

        // ## Runs for anywhere other than Samos Town
        else
        {
            // ## AI movement to approach enemies
            MovementScript.ForcedMove();
        }

        // ## Update Camera Position relative to the player, and allow zoom in and out
        CameraScript.UpdateCamera();
    }
}
```

Figure 4.36: Sample of the **PlayerMainScript** code

The **PlayerMainScript** not only has the centralized Update method, it is also the "hub" of the player Scripts. If one script wants to communicate with another, they fetch the script component from the main script public variables.

The following figure 4.37 summarizes the hierarchy and organization of the player scripts.

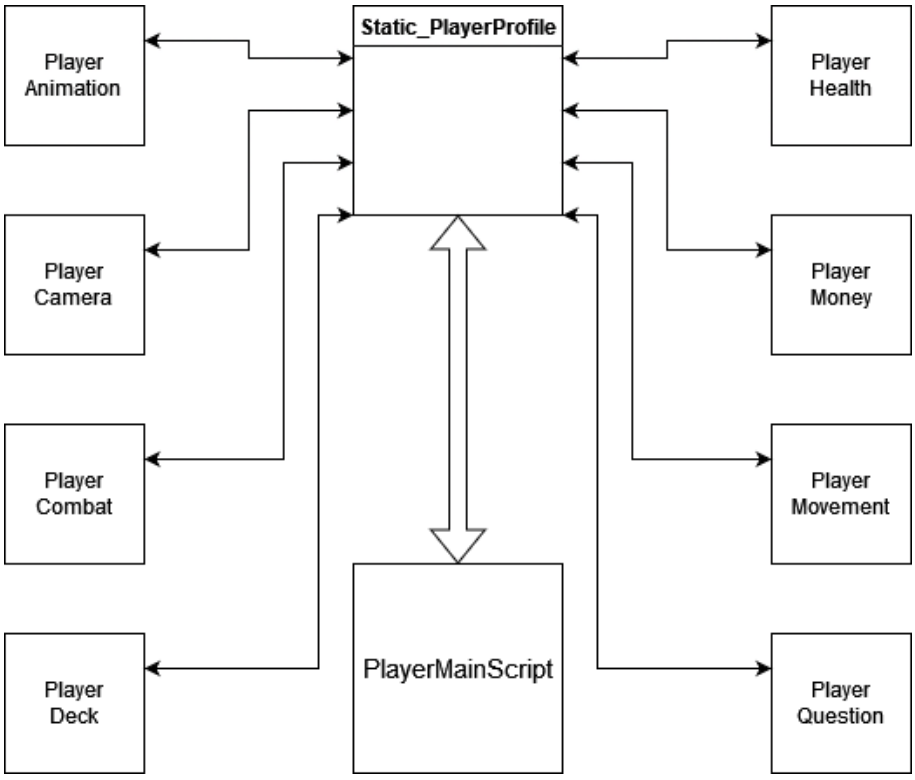


Figure 4.37: Diagram of the player script hierarchy

4.6 HUD interface

Math-Masters contains two separate Canvas: Essential Canvas and Combat Canvas.

The Essential Canvas contains the elements related to the player, while the Combat Canvas contains the elements related to enemies, the question, the hand of cards and the player's health. Essential Canvas is present in Samos Town and in the levels at any given point, on the other hand Combat Canvas is only active during Combat. Due to the nonexistance of combat in Samos Town, the Combat Canvas is not present in that scene. In addition, the Essential Canvas houses the pause menu, and has a HUD button in the shape of a gear that acts the same way as pressing the "ESC" key during gameplay. The figure 4.38 shows the content from the Essential Canvas in-game.

Figure 4.38: Screenshot of the HUD in Samos Town

4.6.1 2D elements in HUD

At the start of each scene with Essential Canvas, the information from the **Static_PlayerProfile** is used to fill the player HUD elements. These elements display the username, current money and the Maximum Level achieved as a smaller number below the displayed username, as seen in the previous figure 4.38.

Like shown in a previous figure 4.16, the combat screen adds more HUD elements. The player health is managed by the **PlayerHealth** script, while the enemy health is managed by the **Enemy** script itself. The lower half of the screen is dedicated to the hand of cards, with a spell counter on the right side.

4.6.2 3D elements in HUD

The player model and enemy model appearing at the corners of the HUD are animated live. This is possible through the use of a dedicated camera with a Culling Mask on a

"UICamera" custom Layer connected to a Render Texture, resulting in only picking up elements in the same layer and leaving the background as transparent. A duplicate player model with its layer set to "UICamera" is placed in front of this camera, and the Render Texture is used as the asset of the Image gameObject placed in the canvas. The process is repeated for the enemy, but in this case the model is deleted and replaced each time a new battle is going to happen. The figure 4.39 shows the set up, which is located out of bounds.

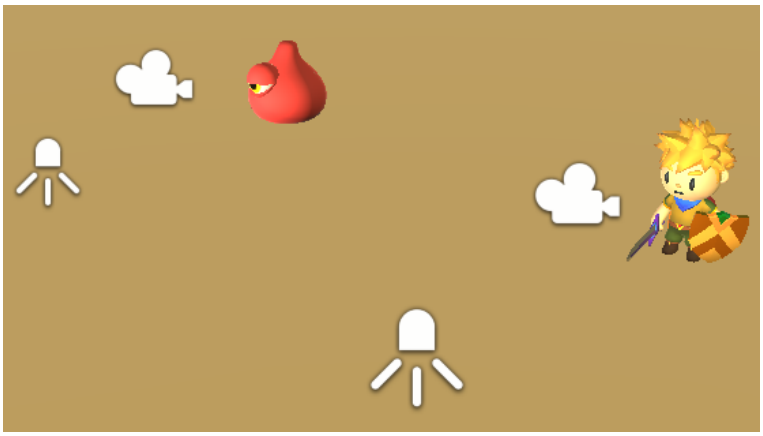


Figure 4.39: Screenshot of the Duplicate models setup for the 3D HUD elements

Chapter 5

Conclusion and Future Work

5.1 Contributions and Achievements

I managed to implement a serious/educational game for the multiplication operation. Even though I wasn't able to test it before delivery, I believe it is the foundation for a fun game that successfully improves the players' basic math skills.

In order to build this game, I had to learn how to use account-based external databases with internet connection to save each players' progress. Not only that, I had to overcome challenges about instantiating assets in the correct places of the canvas. This game is both invested into 3D space and the 2D canvas, so I could not specialize in one in particular. Utilizing ChatGPT boosted my productivity by removing the amount of dead time from searching for solutions (often times finding none). In addition, I was able to get answers from ChatGPT about many programming topics to satisfy my curiosity and desire to expand my knowledge as a programmer.

I like to give autonomy to the designers, so having prepared the game the way it current is made it easy for my colleague to change the content as they see fit. I liked learning/applying unusual visual techniques, such as having animated 3D models appear in the canvas, and adding coding events to the assets' animations to allow a cooperation between animation and coding.

A big difficulty was the constant change of route during the early to mid stages of development. I implemented features to work for a purpose, but in some occasions that

purpose was no longer the main objective. I could not discard the working functions that took so much time, effort and testing. Instead I restricted how much the game could deviate from previously decided objectives.

5.2 Future Work

Math-Masters has several routes to expand upon, either on the educational side or in the genres side. As mentioned in the previous chapter, Samos Town is missing the features that turn it into a Hub. For that, the cosmetic store needs to be implemented, an additional store for player upgrades, such as more Health Points and dealing more Damage to the enemies. This scenes requires NPCs to give it the feel of being a town, who the player can strike conversations with. This idea can be further expanded by having the NPCs be the tutors for the player.

Expanding on the Deck-Building aspect requires the implementation of a customizable deck of cards, more likely to be the SpellCards. The player can purchase new and more powerful SpellCards from the stores at Samos Town, slot them in their deck and gain greater advantages over the enemies in the levels.

The monsters have an EXP public field to grant to the player on death, but there is no EXP mechanic implemented even though the HUD has a dedicated space for a slider. Doing so would add another layer of progression and balancing. Incorporating player levels allows, for example, locking certain upgrades and cards until the players reach a certain level.

Adding alternatives to ways to play with higher difficulty but greater rewards will cater the more competitive players. For example, adding timers that when reach zero, the monster attacks the player.

The game needs a settings menu, to adjust music volume, entity volume, reset player progression, rebind keys, among other options. This greatly improves accessibility. In addition, implement the "About Me" button by displaying slides showing who are the developers and what is this game's purpose.

Expand on the database. By adding more elements stored in database, teachers can overview the progress of their students. These elements can range from how much time is spent playing, to how many times a level is completed. For this to happen, a new type of account has to be created to show all the relevant information of the class to the teacher in-game. This requires implementing an optional grouping method, joining student accounts according to their class.

More importantly, this game needs to be tested by the targeted audience. The future testing will be made alongside a form so as to assess and acknowledge what are the good points and the bad points. With the data, we will change the game by strengthening the good and altering/removing the bad.

Bibliography

[1] Undertale. Toby fox. https://undertale.com, September 2015.

[2] Unity Engine. Unity. https://unity.com/, June 2005.

[3] Frutuoso G. M. Silva. Practical methodology for the design of educational serious games. *Information*, 11(1), 2020.

[4] André Barbosa, Pedro Pereira, João Dias, and Frutuoso Silva. A new methodology of design and development of serious games. *International Journal of Computer Games Technology*, 2014:1–8, 06 2014.

[5] Fun Math Facts: Games for Kids. Speedymind llc. https://play.google.com/store/apps/details?id=net.speedymind.mental.arithmetic.trainer.learning.games.practice.k5.grade.math.vs.slimes&gl=us, October 2020.

[6] Prodigy Math. Prodigy education inc. https://play.google.com/store/apps/details?id=com.prodigygame.prodigy, May 2018.

[7] Void Tyrant. Armor games studios. https://armorgamesstudios.com/games/void-tyrant, June 2019.

[8] Slay the Spire. Mega crit games. https://store.steampowered.com/app/646570/Slay_the_Spire/, January 2019.

[9] TUNIC. Tunic team. https://tunicgame.com/, March 2022.

[10] Unity Asset Store. Unity. https://assetstore.unity.com/, November 2010.

[11] Unreal Engine. Epic games. https://www.unrealengine.com/en-US, May 1998.

[12] Epic Games. Epic games store. https://store.epicgames.com/en-US/, December 2018.

[13] PlayFab. Microsoft. https://playfab.com/, January 2014.

[14] Unity3d PlayFab SDK. Microsoft. https://learn.microsoft.com/en-us/gaming/playfab/sdks/unity3d/, March 2015.

[15] ChatGPT. Openai. https://chat.openai.com, June 2020.

[16] TextMeshPro. Digital native studios. https://docs.unity3d.com/Manual/com.unity.textmeshpro.html, 2014.

.