HELLENIC MEDITERRANEAN UNIVERSITY

Department of Electrical and Computer Engineering

Study Program: Informatics Engineering

THESIS

Design and Implementation of turn-based card game in Unity

Σχεδιασμός και Υλοποίηση turn-based card game στη Unity

SARIDAKIS ANASTASIOS AM3967

ADVISOR

IOANNIS PACHOULAKIS

MARCH 2021

Acknowledgements

I would like to thank my thesis advisor, Dr. Pachoulakis Ioannis, for his insightful and helpful recommendations during the development of this paper. His willingness to freely offer his time is appreciated.

I would also like to express my gratitude to my professors and colleagues at the Hellenic Mediterranean University, who have assisted me in learning much of what I now know about game development and patiently encouraged me.

Most of all, I would like to express my gratitude to my parents and family for their unwavering support over the years; without them, I would not be where I am today.

Περίληψη
Στόχος της πτυχιακής εργασίας ήταν η δημιουργία ενός δυσδιάστατου παιχνιδιού πολλών παικτών με ψηφιακές κάρτες,.

Οι παίκτες έχουν την δυνατότητα να μπορούν να συνδεθούν στο παιχνίδι με τον δικό τους προσωπικό λογαριασμό και να δημιουργήσουν της δικές τους τράπουλες, επιλέγοντας κάρτες από την συλλογή τους. Οι τράπουλες είναι περιορισμένου μεγέθους, όπου κάθε κάρτα έχει τα δικά της χαρακτηριστικά όπως κόστος, ζημιά και ζωή. Επομένως, οι παίκτες πρέπει να χρησιμοποιήσουν στρατηγική κατά την δημιουργία της τράπουλας τους.

Έπειτα, ο κάθε παίκτης μπορεί να μονομαχήσει διαδικτυακά με οποιοδήποτε άλλο παίκτη επιθυμεί. Η μονομαχίες αποτελούν τον πυρήνα του παιχνιδιού, καθώς ο κάθε παίχτης θα πρέπει στον γύρο του να πάρει αποφάσεις. Οι παίκτες ανταλλάζουν γύρους έως ότου κάποιος παίκτης χάσει.

Κατά την δημιουργία της πτυχιακής χρησιμοποιήθηκε η μηχανή παιχνιδιών Unity3D, καθώς και τεχνολογίες δικτύωσης όπως το Mirror. Τέλος χρησιμοποιήθηκε η υπηρεσία της Google - Firebase, η οποία μας παρέχει τη βάση δεδομένων όπου αποθηκεύεται το state του παιχνιδιού.

Abstract

The aim of the present thesis was to create a two-dimensional multi-player digital card game.

Players have the ability to log into the game with their own personal account and create their own deck of cards by selecting cards from their collection. The decks are limited in size and each card has properties such as cost, damage and life points. As a result, when building their deck, players must utilize strategy.

After building their decks, players are able to duel online. These duels are the game's core logic, where each player must analyze and make decisions in turn. The players alternate rounds until one loses.

The thesis was developed using the Unity3D gaming engine, with the help of network technologies such as Mirror. In addition, Google Firebase service offers a database to store online data.

Table of Contents

1. Introduction 9
    1.1 Purpose of the project 9
    1.2 Where the game originated from 9
    1.3 Objectives of the game 9
2. Technologies that were used 10
    2.1 Game Development 10
    2.2 What is a game engine? 11
    2.3 Why Unity? 11
        2.3.1 What is Mirror 11
    2.4 What are Databases and why Firebase. 11
3 The Game 13
    3.1 The Player Experience Graph 13
    3.2 The parts of the game 14
        3.2.1 The registration scene 14
        3.2.2 The main menu scene 15
        3.2.3 The battlegrounds scene 19
    3.3 The ruleset and the mechanics of the Duels 20
        3.3.1 The cards 20
        3.3.2 The H.U.D. and the Drop zones 21
        3.3.3 Mulligan 22
        3.3.4 Turns 23
        3.3.5 Adding Clarity 23
        3.3.6 Animations and sounds 24
4.The code implementation and the Classes that were used. 26
    4.1 Registering and Signing In 26
        4.1.1 Registration Panel 26
        4.1.2 Login Panel 29
    4.2 The implementation of the main menu 30
        4.2.1 UIManager 31
        4.2.2 DataBridge 32
    4.3 Deck Manager 35

4.4 Scriptale Objects and Cards. 35
4.4.1 Cards Instantiation 35
4.4.2 DropZones 38
4.4.3 Card movement 39
4.4 GameManager 42
4.5 Mirror implementation 45
4.5.1 Authority 45
4.5.2 Attributes 45
4.5.3 Remote Procedure Calls 46
4.5.4 PlayerManager 46
5. Network Infrastructure – Firebase Services 53
5.1 Firebase Authentication 53
5.2 Realtime Database 54
5.2.1 The structure of the Realtime Database 55
6. Conclusion 57
6.1 Problems faced during development 57
6.2 Improvements for the future 57
6.3 Personal fulfillment 58
Bibliography and Sources 59

Table of Figures

Figure 1: Time Consumed Per Development Stage 10
Figure 2: Player Experience Graph 13
Figure 3: Sign-in Screen 14
Figure 4: Register Screen 15
Figure 5: Main Menu 16
Figure 6: Options Panel 16
Figure 7: Deck Selection Panel 17
Figure 8: Deck Manager 18
Figure 9: Choose Card to Play Panel 19
Figure 10: In-game Battleground 20
Figure 11: Card Analysis 21
Figure 12: Friendly (green) and enemy's (red) DropZones. 22
Figure 13: Mulligan phase 23
Figure 14: Card Shader Effects 24
Figure 15: Dissolve Card Effect 24
Figure 16: Sound Mixers 25
Figure 17: Verify Input Script 26
Figure 18: Register Code 27
Figure 19: Register in Firebase 27
Figure 20: Database Structure 28
Figure 21: Switch between Sign-in and Register 28
Figure 22: Firebase Connection to the Game 29
Figure 23: Login Script 29
Figure 24: Load on login - Enter Main Menu 30
Figure 25: Main Menu Hierarchy 30
Figure 26: UIManager in Inspector 31
Figure 27: Load Decks from Firebase 32
Figure 28: Making DataBridge Singleton 32
Figure 29: Load decks from Firebase 33
Figure 30: Save deck to Firebase 34
Figure 31: Deck Manager Inspector 35
Figure 32: Card Class 36
Figure 33: Creating Cards from Menu 36
Figure 34: Scriptable Objects of type Card 37
Figure 35: Card Database 37
Figure 36: Card Display Script 38
Figure 37: Dropzones Script 39
Figure 38: Dropzones Script part 2 39
Figure 39: Draggable Script 40
Figure 40: Draggable Script part 2 41
Figure 41: Draggable Script Part 3 42
Figure 42: GameManager Script 43
Figure 43: Enums 43

Figure 44: GameManager script part 2 44
Figure 45: End Game Clauses 44
Figure 46: PlayerManager Script 46
Figure 47: Initialize players turn 47
Figure 48: DealCards script 47
Figure 49: Command for Mulligan 48
Figure 50: Rpc to show cards in Mulligan 49
Figure 51: Rpc to show cards in hand 50
Figure 52: Rpc to play card 51
Figure 53: Rpc change turn 52
Figure 54: Firebase Authentication Sign-in methods 53
Figure 55: Authentication in Firebase 54
Figure 56: Realtime Database 54
Figure 57: RealTime Database structure 55
Figure 58: Cards ID's in Database 56

1. Introduction

1.1 Purpose of the project

The aim of this thesis was to build my own game from scratch. My passion being game development, this was my opportunity to create a game that expresses me. I adopted several ideas from other games and added my own twists to them. The end result is a 2D multiplayer card game in which players build unique decks from a variety of available cards and compete against each another.

1.2 Origins of the game

The game's concept was inspired by Blizzard's successful free-to-play digital collectible card game Hearthstone. In the game, players engage in 1-on-1 battle against other players or AI opponents, attempting to eliminate the opposing hero before being destroyed themselves. Players may accomplish this by employing a range of spells and minions (creatures sent into the battlefield to fight on their behalf), as well as equipping weapons and jumping right into the action. Hearthstone was created to be a simple, enjoyable, and accessible game, but it also has a lot of depth, strategy, and complexity for more advanced players. In April 2016, it was revealed that the game has exceeded 50 million registered users.[1]

1.3 Objectives of the game

The goal of the game is to defeat other players in one-on-one situations by using a custom deck that each player created.

The game's strategy is divided into two parts. The first involves proactive strategic thinking, and it occurs when a player constructs his own unique decks from a collection of cards. Each card has its unique set of values, including cost, life, and damage. Because each deck can contain up to 30 cards, the player must think twice before adding it to his own deck. In a sense, a player must find the right balance between cost and stats for his deck.

When players duel, they must think strategically in a reactive manner. The players must make decisions as to which cards to play and when to play them. Players have the option of reacting to the cards dealt by their opponents. They must deduce the motivations of their opponents and plan. However, as with most games, the ultimate goal of the game is to have fun.

2. Technologies that were used

2.1 Game Development

Game development, as the name suggests, is the art of creating games. Some stages of game development are the design of said game, the production of it and finally its publication.

In the design stage, the game designers will plan out how the game will play. That includes every decision from the start to the end of the game. Moreover, they will try to answer questions such as for what audience the game appeals to and how long will it take for it to be actually developed. Finally, they will come up with some type of prototype of the game, to see if it works in practice. Typically, that takes about 1/5th of the total production time.

Production is the stage that takes the longest. In this stage the developers create the characters, the environment, the dialogs, the sounds, the effects, etc. They constantly try out everything to see if it fits together. A big chunk of their work usually does not even make it to the end, as they deem it does not fit the gameplay. They constantly change things up for it to feel more "fluid".

Last is the publication of the game. In this stage the game's content is ready (or at the very least "almost" ready). Usually the AAA companies (companies with huge budget and number of employees) have a publisher. That means a company that specializes in the marketing and distribution in exchange for a portion of the profits. If the company is small and cannot afford a publisher, they publish the game themselves.

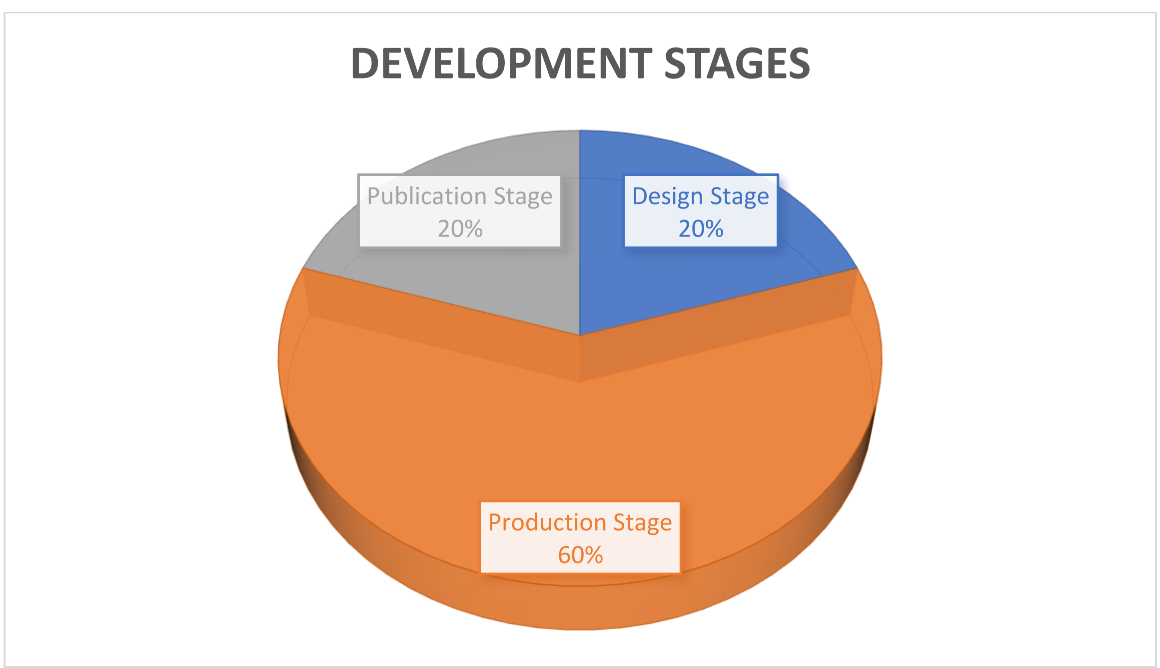

Figure 1: Time Consumed Per Development Stage

2.2 What is a game engine?

A game engine is the core software necessary for a game to properly.[2] Game engines usually provide a selection of development tools and reusable components that are needed for a game to be created. These engines constantly take care of things such as rendering the graphics, animations, artificial intelligence and many more things. Because its something so vital, its important for game developments to choose the right game engine that suits their needs. That's why big developer studios create their own game engines from scratch. That enables them to choose exactly what features the engine will contain and how it will work. That's a long and expensive process however. Smaller studios that don't have the budget and/or time to create their own game engine, use pre-existing ones. Such engines are Unity and Unreal Engine.

2.3 Why Unity?

Unity is a cross-platform game engine.[3] It can be used to make simulation games in 2D, 3D, virtual and augmented reality. Some of its benefits are its user friendly interface, it can support a vast array of platforms from desktop to mobiles, it has a great community with extra tools that are great and lastly its free.

Unreal Engine, another game engine such as Unity, also provides many benefits. Such benefits are, it can handle a wide variety of games, better inherently shooting and first person mechanics overall. It also supports cross-platform games like Unity does and lastly it is also free.

For the purposes of this project I went with Unity as I am making a 2D card game and not a shooter. Moreover the friendliness of Unity's UI gives it an advantage.

2.3.1 What is Mirror

Mirror is a Unity networking library that works with a variety of low-level transports.[4] Mirror, at its core, is a technology that adds multiplayer functionality to Unity games. It is made up of a lower-level transport real-time communication layer that handles a lot of the activities that multiplayer games require. Mirror prioritizes usability and iterative development, and it offers important features right away.

2.4 What are Databases and why Firebase.

A database is a collection of data that is organized and kept in a computer system and can be accessed electronically.[5]

A cloud database is a database that runs on a cloud computing platform and provides as-a-service access to the database. Users can run databases on the cloud themselves using a virtual machine image, or they can pay for access to a database service provided by a cloud database provider. The underlying software stack is transparent to the user thanks to database services.[6] A service like that is Firebase's Realtime Database.

Firebase is a service that provides developers with a vast array of tools. These tools provide solutions to common game development “issues” that occur during development stages, thus giving the developers the freedom to focus on the app experience itself. Tools that are included in the firebase ecosystem are cover areas of analytics, authentication, databases configuration, file storage etc.[7]

3 The Game

3.1 The Player Experience Graph

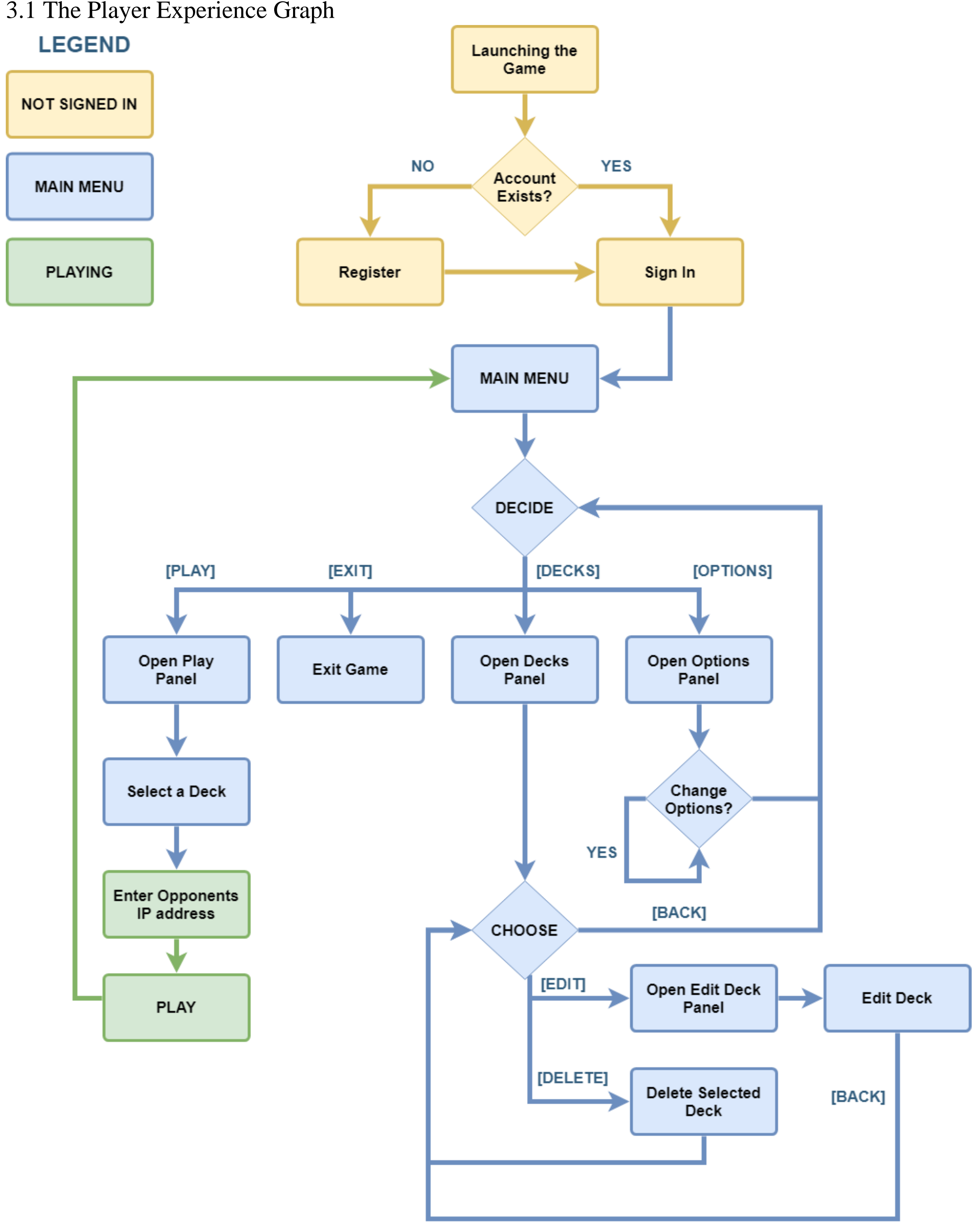

Figure 2: Player Experience Graph

3.2 The parts of the game

3.2.1 The registration scene

The game consists of three primary scenes. The first is the registration scene where the user would have to enter his credentials. In order to access the game an account is necessary. That's because the game is online and each player can create different decks. That means we would have to do two essential things. Firstly, we would have to store each player's decks and secondly these decks must be accessible at any time across any platform or system. Therefore, we solve both of these issues by implementing an account system.

When first opening the game, the player is presented with a choice to either sign in with his credentials or to register a new account.

Figure 3: Sign-in Screen

If the player chooses to register, then an email is required as well as a username and a password. After the player selects register he would have to go back to the sign in screen and connect using his credentials.

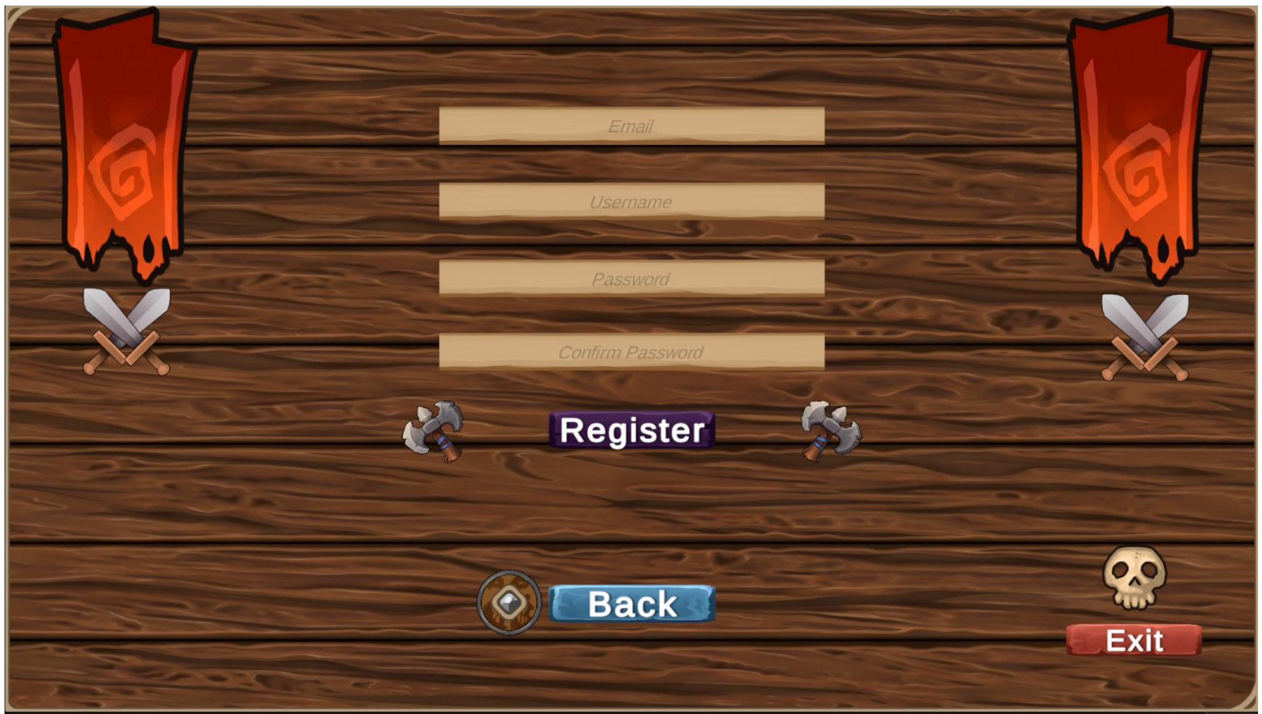

Figure 4: Register Screen

Only his email and password are required for him to connect. After entering his credentials he will enter the main menu of the game.

3.2.2 The main menu scene

The main menu is the second of the three primary scenes in our project. The main menu consists of four buttons. The last of them is the Quit button where a player can exit the game.

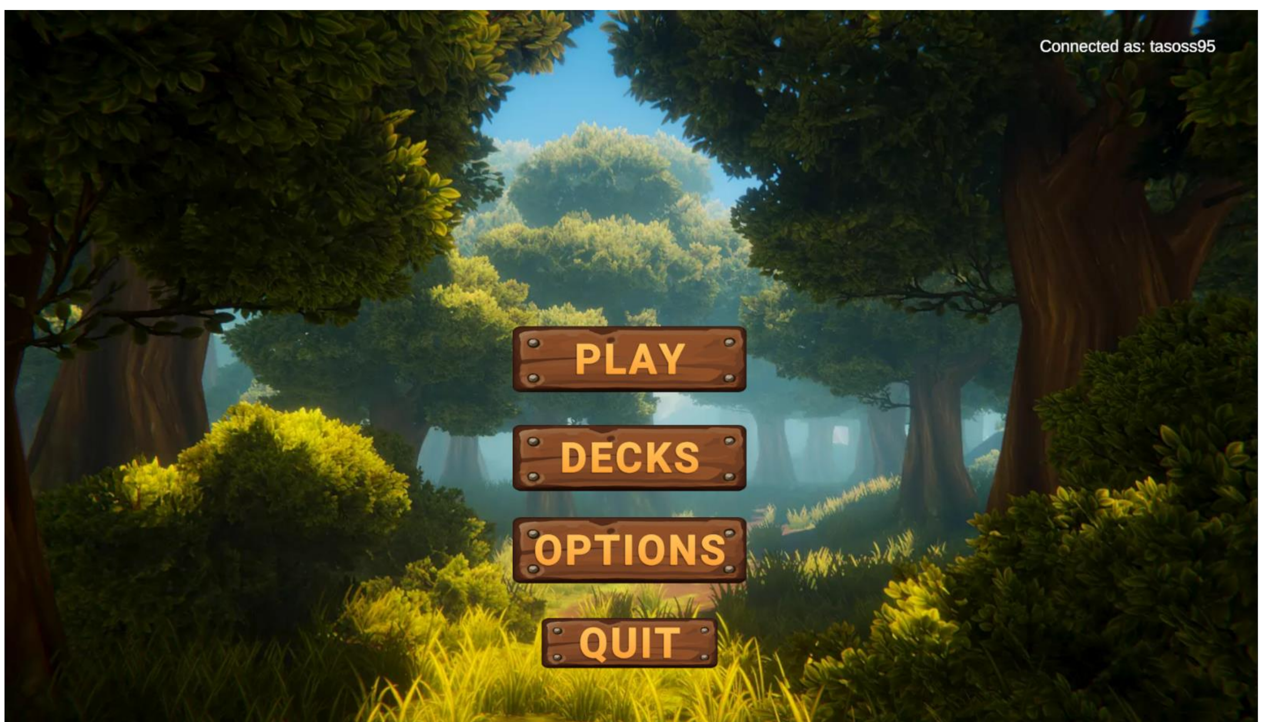

Figure 5: Main Menu

In the Options button a player can change the games volume, the quality and resolution settings.

Figure 6: Options Panel

The second button is the Decks button, where a player can create, edit or delete a deck. When the player opens the deck window he will be presented with all of his created decks up to 8. Those

decks will be loaded directly from the database (which I will analyze in the next chapter). Selecting a deck will let the player either delete or edit the particular deck.

Figure 7: Deck Selection Panel

If the player chooses to edit the deck or create a new one then the Deck Creation panel opens. There, the game will load the already selected deck if the player chose the edit option. Otherwise, if the player had selected to create a new deck, then the game wouldn't load a deck. In any case, the player can drag and drop any card he wishes to use in his deck.

Figure 8: Deck Manager

The Deck Creation window is simple to use. On the right side the player is presented with every possible card the game has to offer, ordered by value of cost. On the left side the player can see in a more minimized view, his current deck. On the upper left corner it displays the card deck's name.

Each deck must have exactly 30 cards. A card count number exists on the bottom right corner to display this. The player can add or remove cards by simply dragging the card he wishes to add to his deck from the right side to the left and vice versa for removing them. Once the player is ready to save the deck he simply exits the window and a pop up message informs him that his deck will be saved.

Last but not least, the player can click on the play button and a Deck Selection window opens up. There the player must choose the deck he will be playing with and once he has, a play button will lighten up.

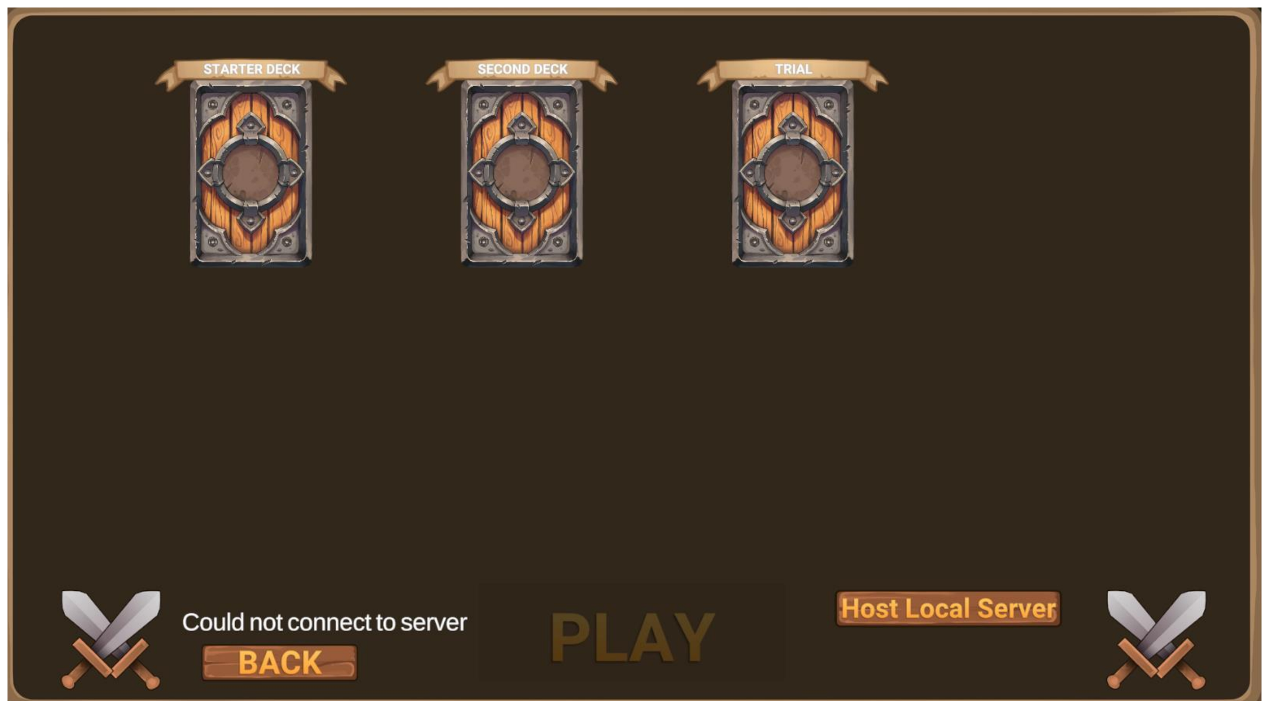

Figure 9: Choose Card to Play Panel

3.2.3 The battlegrounds scene

This is arguably the most important scene of the game. Here the players duel each other.

Firstly, a player takes the role of the host and the other player takes the role of a client. The client has to enter the IP address of the host. The client will connect and stay connected with him as long as the host exists. When both players hit the Ready button the game will begin. The players duel each other until either of their health points reaches 0. When this happens the player with the 0 health points loses and the other player wins. After they finish their duel, the players get returned to the main menu.

Figure 10: In-game Battleground

3.3 The ruleset and the mechanics of the Duels

In this part of the thesis, I will explain all the different mechanics and rules that take place during a duel between players.

3.3.1 The cards

The most important aspect in a card game, are of course the cards themselves. Each card displays on the upper left corner their cost in mana crystals, in the bottom left corner their attack points and in the bottom right corner their life points. The card's name can also be seen in the top area and in the center area an artwork of the card exists. Bellow the artwork there is some space for a description (although I'm not using it yet). I purchased the cards artwork from the asset store[8], but I made the card borders and the icons myself.

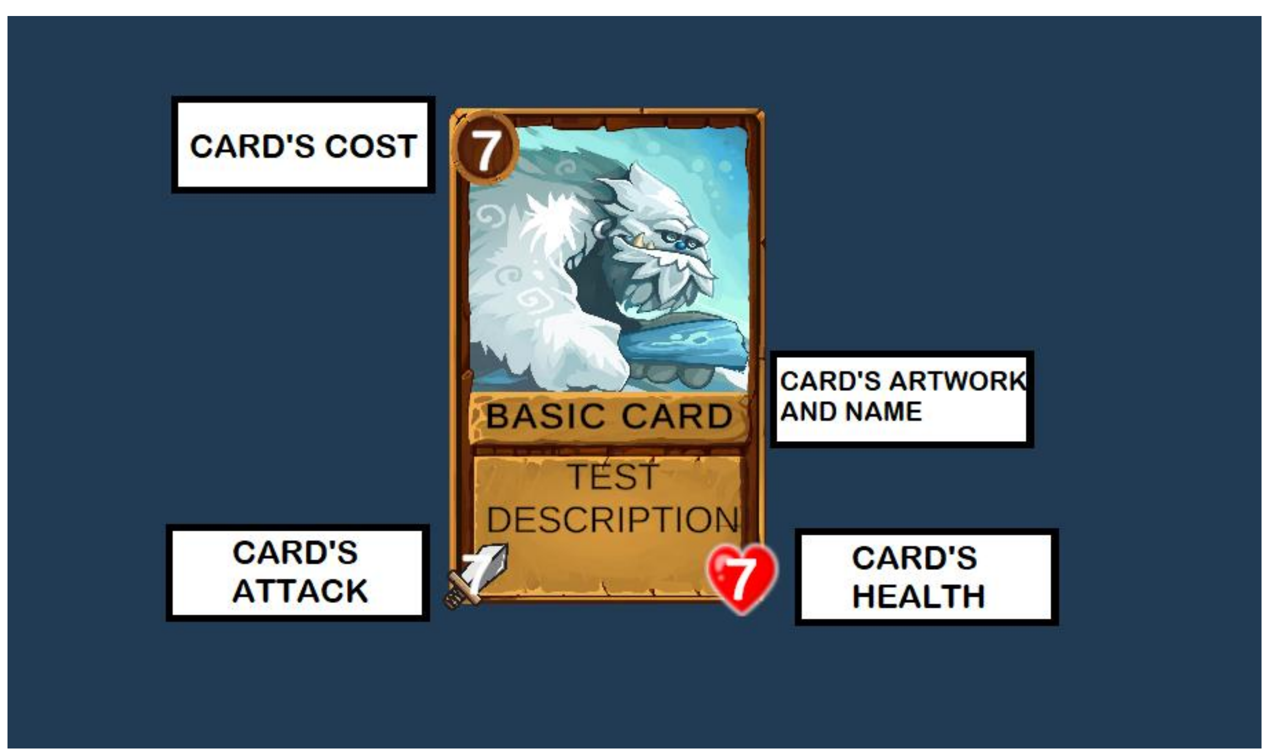

Figure 11: Card Analysis

3.3.2 The H.U.D. and the Drop zones

During the duel, the players H.U.D. (Heads Up Display) gets engaged. Each player can see their avatar on the bottom left corner and their opponent's avatar on the upper right corner. Bellow each player's avatar resides their remaining life HP (Hit Points). The player's Mana Crystals are shown on the bottom right corner. Each time a player gets a turn their Mana Crystals are refunded, and their cap is increased by 1 up to 10. That means Players will start with 1 mana in turn 1 and have 10 mana by turn 10. Every player can see their deck above their Mana Crystals. Players can also see how many cards their opponent holds in their hand, albeit they cannot see what cards they are.

Drop zones are the area's a player (or the game) can drop cards. Each player has their hand area and their respective drop zone. A player cannot drop a card in the opponents drop zone or hand.

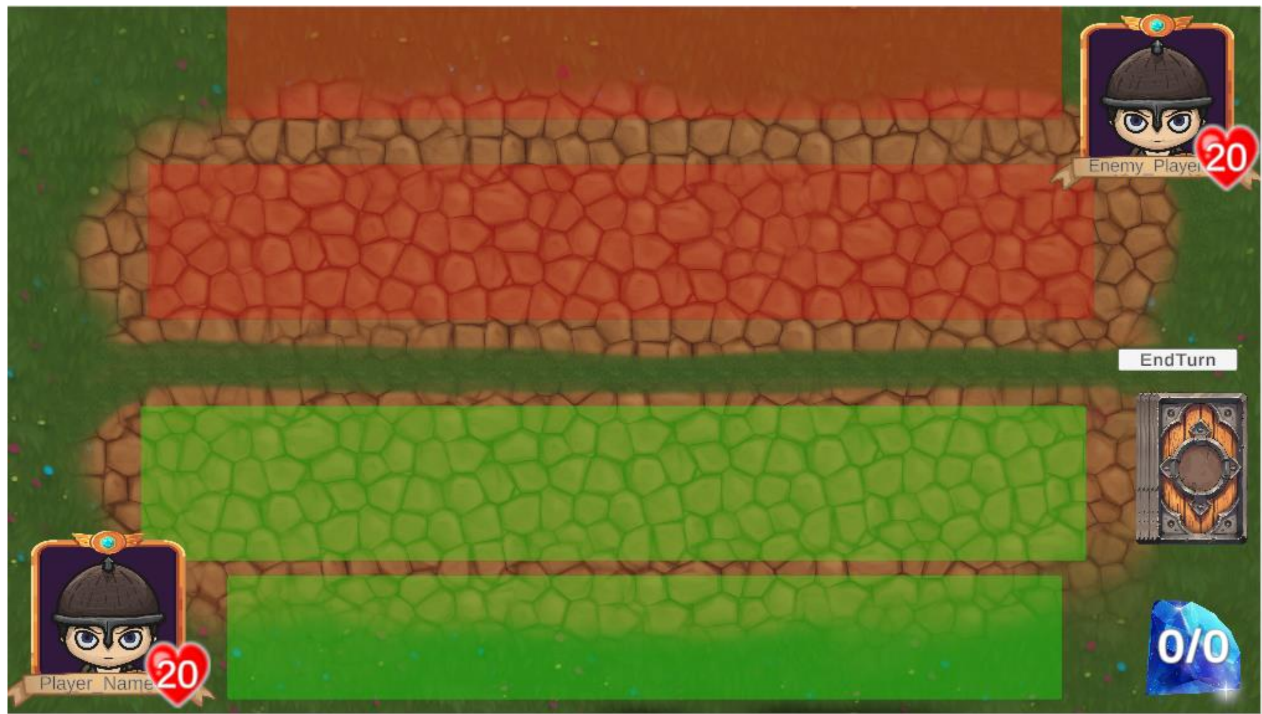

Figure 12: Friendly (green) and enemy's (red) DropZones.

3.3.3 Mulligan

When the duel begins, each player will draw 5 random cards from their deck. The players can either choose to keep them or to Mulligan. Mulligan means that they will discard their 5 cards back in their deck and draw 5 new random cards. Each player can only Mulligan once and only at this stage, meaning that after they mulligan they have to keep them.

Figure 13: Mulligan phase

3.3.4 Turns

For a player to take any action, it has to be his turn. Always the player that acts as the host starts first. At the start of each turn, the player's Mana Crystals refill and they draw a card from their deck. During their turns player can drop cards on to the battlefield and/or attack enemy cards or even the enemy player itself.

For a player to drop a new card to the battlefield, they must have equal or more mana crystals than the cost of card they want to drop. Players gain Mana Crystals passively as the game progresses, up to 10. When the player drops a card, the cost of the card is withdrawn from their available Mana Crystals.

After a card is dropped the player must give his opponent the chance to "react". That means that he cannot attack with his card unless a turn has passed since he played it. Once a turn has passed, the card is ready to attack. The player has to select it and then select a target.

A target can be anything from the opponents dropped cards to the opponent's avatar itself.

Once a player is happy with his actions, he must press the "End Turn" button to pass the turn to the other player.

The game goes back and forth until a player loses all of his life Hit Points.

3.3.5 Adding Clarity

Visual clarity is one of the most important things a game developer can add to their game. It's imperative for a player to be able to see all of his available actions in a single glance. In order to help the players keep track which of their cards can attack I added a visual indication around them.

If a card is available to attack, it will be surrounded by green flames. When the player selects a card to attack with, the fire of that card will turn blue. Subsequently as the player hovers with his cursor over the enemy cards a red flame appears to surround them indicating that they are a target. Moreover, the cursor turns from a hand to a sword icon.

Figure 14: Card Shader Effects

3.3.6 Animations and sounds

To expand in the matter of clarity, animation and sound cues are also used to indicate some specific situations. In particular, when a card attacks another card an animation is played letting both players know what's happening. Additionally, when a card is destroyed it emits a special "dissolving" shader designating its destruction.

Figure 15: Dissolve Card Effect

In the scope of sound, I have added a Master volume that controls the general volume and two more controls, one for the music and the other for the sound effects such as button clicks and attack ques. Unity makes controlling the sound really easy with the use of mixers.

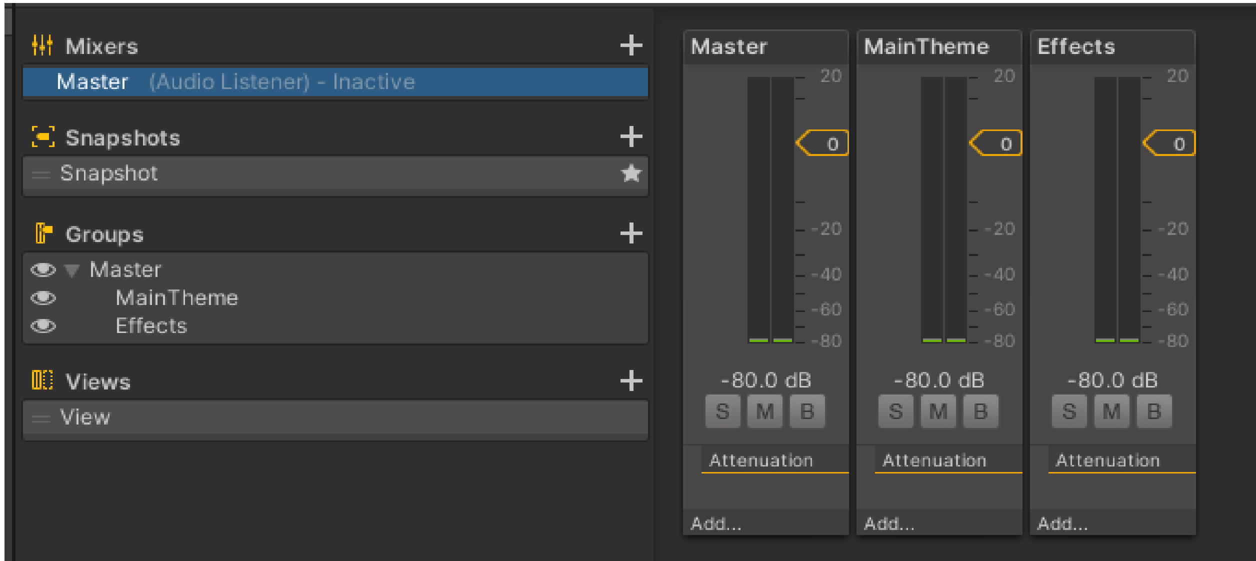

Figure 16: Sound Mixers

Here I have the MainTheme (music) mixer and the effects mixer. Both of them pass through the Master mixer. That way I can control separately which sounds I want to lower-raise will still keeping the general volume control.

4.The code implementation and the Classes that were used.

In this sector of my thesis, I will analyze and explain in depth the code that I used. I will delve deeper in order to explain the reasons behind my decisions.

4.1 Registering and Signing In

In order to create the sign in and registration screen I have placed a background and two panels. The first panel contains the login buttons and form. The second panel contains the registration form.

4.1.1 Registration Panel

So, to switch between the two panels, I have created a script called "Registration"

```
public void VerifyInput()
{
    if (usernameField.text.Length >10 || usernameField.text.Length<4)
    {
        errorLabel.SetText("Username must be between 4-10 characters");
        submitRegisterBtn.interactable = false;
    }
    else if(passwordField.text.Length < 8)
    {
        errorLabel.SetText("Password must be more than 8 characters");
        submitRegisterBtn.interactable = false;
    }
    else if (passwordField.text != confirmPwField.text)
    {
        errorLabel.SetText("Passwords must match");
        submitRegisterBtn.interactable = false;
    }
    else
    {
        errorLabel.SetText("");
        username = usernameField.text;
        submitRegisterBtn.interactable = true;
    }
}
```

Figure 17: Verify Input Script

In that script a method named VerifyInput() checks that all the parameters are satisfied. In case an input is not correct, a warning message will appear to the user.

Then, Register() will logout the user from his current session (if he is in one) and sends the information to firebase to save his credentials.

```
public void Register()
{
    if(emailRegInput.text.Equals("") && passwordRegInput.text.Equals(""))
    {
        print("Please enter an email and password to register");
        return;
    }
    Logout();
    password = passwordRegInput.text;
    FirebaseAuth.DefaultInstance.CreateUserWithEmailAndPasswordAsync(emailRegInput.text, password).ContinueWith((task =>
    {
```

Figure 18: Register Code

When this method is completed the following will happen.

```
if (task.IsCompleted)
{
    print("Registration Completed"); //from here try to save into firebase, username Pid and deck.
    Player data = new Player(username, FirebaseAuth.DefaultInstance.CurrentUser.UserId, password);
    Deck data2 = new Deck(decks[0].DeckName, decks[0].PlayerDeck);
    Deck data3 = new Deck(decks[1].DeckName, decks[1].PlayerDeck);
    string jsonData = JsonUtility.ToJson(data);
    string jsonData2 = JsonUtility.ToJson(data2);
    string jsonData3 = JsonUtility.ToJson(data3);
    //databaseReference.Child("Users").Child(username).SetRawJsonValueAsync(jsonData);
    databaseReference.Child("Users").Child(FirebaseAuth.DefaultInstance.CurrentUser.UserId).SetRawJsonValueAsync(jsonData);
    databaseReference.Child("Users").Child(FirebaseAuth.DefaultInstance.CurrentUser.UserId).Child("Decks").Child("0").SetRawJson
    databaseReference.Child("Users").Child(FirebaseAuth.DefaultInstance.CurrentUser.UserId).Child("Decks").Child("1").SetRawJson

    gameObject.GetComponent<Registration>().SwitchPanel();
}
```

Figure 19: Register in Firebase

I have created a player variable called data. Player is a class that contains fields for the players username, ID and password. Then I will create 2 more variables of type Deck. These data2 and data3 variables contains the information about the two decks that each player starts with. Afterwards, I encode these 3 data variables to Json format. Then I get the referenced database and I create the path which will structure the database.

Currently, the format I save the data is:

Realtime Database

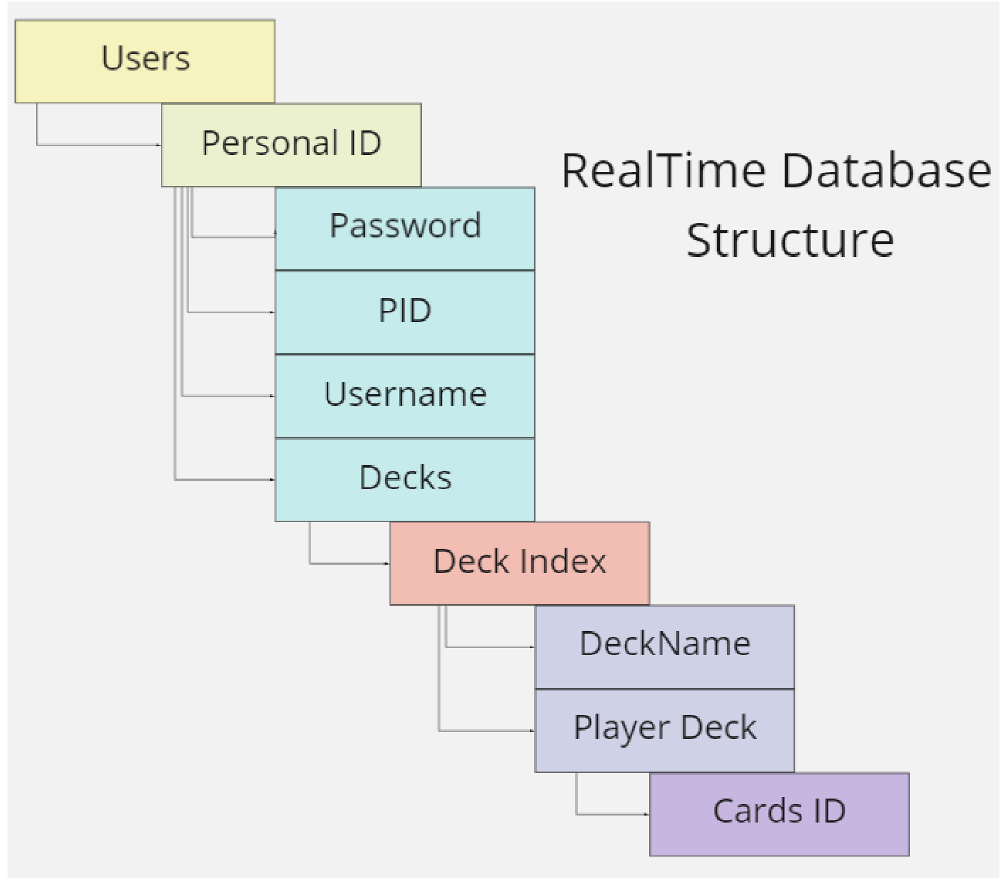

Figure 20: Database Structure

```
public void SwitchPanel()
{
    emailField.text = "";
    authC.SetUsername(username);
    usernameField.text = "";
    passwordField.text = "";
    confirmPwField.text = "";
    errorLabel.text = "";
    logginEmail.text = "";
    logginPassword.text = "";
    loginErrorLabel.text = "";
    gameObject.GetComponent<AuthController>().errorTextLoggin = "";
    gameObject.GetComponent<AuthController>().errorTextRegister = "";
    loginPanel.gameObject.SetActive(!loginPanel.gameObject.activeSelf);
    registerPanel.gameObject.SetActive(!registerPanel.gameObject.activeSelf);
}
```

Figure 21: Switch between Sign-in and Register

Finally, when the user clicks the Register button, the SwitchPanel method will take care to erase the form and switch back to the Login Panel.

4.1.2 Login Panel

To start with, I must establish a connection with the database. Specifically Google's Firebase Service.

```
private string DATA_URL = "h[redacted]/";
```

```
private void Start()
{
    databaseReference = FirebaseDatabase.GetInstance(DATA_URL).RootReference;
```

Figure 22: Firebase Connection to the Game

DatabaseReference gets the database URL and connects the instance of the game with the database.

```
public void Login()
{
    FirebaseAuth.DefaultInstance.SignInWithEmailAndPasswordAsync(emailInput.text, passwordInput.text).ContinueWith((task =>
    {
        Debug.Log("Password Log is: " + passwordInput.text);
        if (task.IsCanceled)
        {
            Firebase.FirebaseException e = task.Exception.Flatten().InnerExceptions[0] as Firebase.FirebaseException;

            GetErrorMessage((AuthError)e.ErrorCode, "Loggin");
            return;
        }
        if (task.IsFaulted)
        {
            Firebase.FirebaseException e = task.Exception.Flatten().InnerExceptions[0] as Firebase.FirebaseException;

            GetErrorMessage((AuthError)e.ErrorCode, "Loggin");
            return;
        }
        if (task.IsCompleted)
        {
            print("User logged in");
            LoadNextScene = true;
        }
    }));
}
```

Figure 23: Login Script

Then when the user enters his credentials I use the asynchronous method

FirebaseAuth.DefaultInstance.SignInWithEmailAndPasswordAsync. What this method does, is that it sends a packet with the users password and username and firebase checks if its correct.

If it is then I turn the Boolean variable LoadNextScene to true. If its not, I print the error message to the user.

```
private void Update()
{
    if(errorTextLoggin != "")
    {
        LoginLabel.SetText(errorTextLoggin);
    }

    if (errorTextLoggin != "")
    {
        LoginLabel.SetText(errorTextLoggin);
    }

    if (LoadNextScene)
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
    }
}
```

Figure 24: Load on login - Enter Main Menu

In the Update method, which is called every frame, when LoadNextScene becomes true, I load the next main menu. I check it every frame, in case some delay takes place.

4.2 The implementation of the main menu

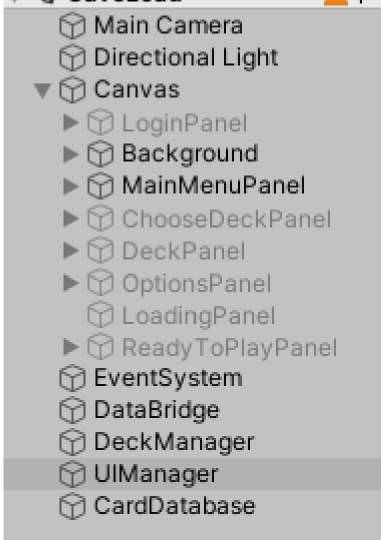

Figure 25: Main Menu Hierarchy

Every element of the main menu has a parent panel. With that in mind, whenever a user clicks a button the respective panel opens. The most important of them being the UIManager, the DataBridge and the DecksPanel.

4.2.1 UIManager

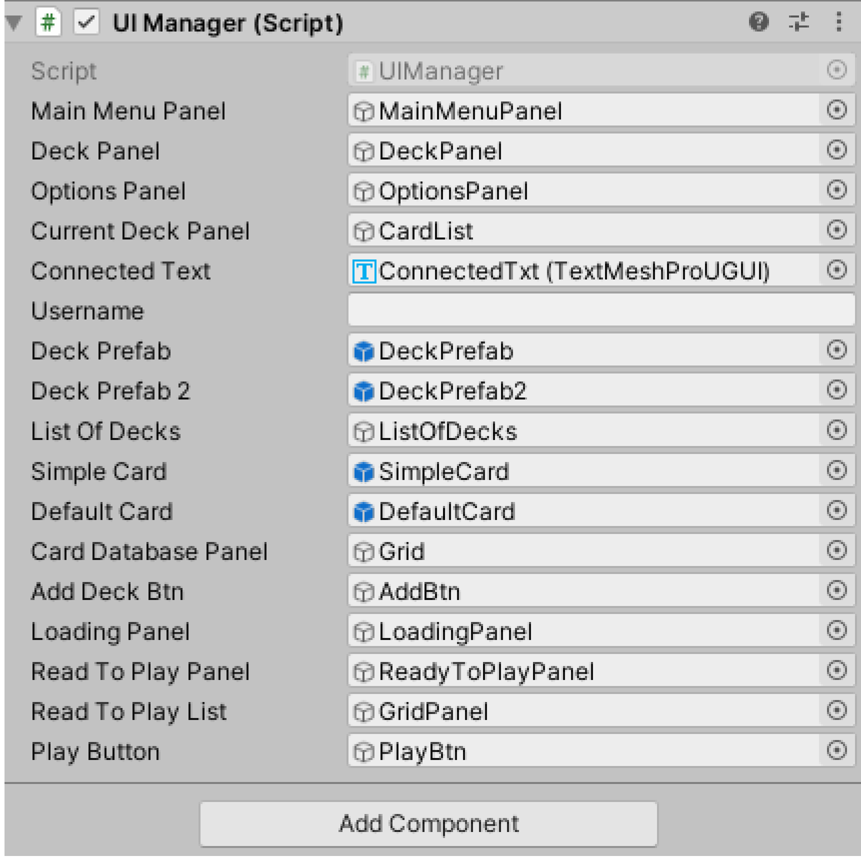

Figure 26: UIManager in Inspector

Due to the complexity of the numerous elements and the interaction between them, a manager class is very important. It will help to trivialize the interactions between the panels as they open and close. Moreover, all the references are in one place, instead of being scattered all over the place.

That being said, most of the variables are public, so any other class inside the project can use them, without having to reassign them.

One of the most important methods inside the UIManager, is the LoadDecks(). Its called whenever the player opens the deck panel and its responsible of loading the decks from the DeckManager to the panel.

```
public void LoadDecks()
{
    if (listOfDecks.transform.childCount != 0)//Clear the decks so i can re-instantiate them
    {
        print("Im inside !=0");
        if(listOfDecks.transform.childCount< DeckManager.Instance.PlayerDeckList.Count)
        {
            print("Im inside listOfDecks.transform.childCount< DeckManager.Instance.PlayerDeckList.Count");
            for (int i = listOfDecks.transform.childCount - 1; i < DeckManager.Instance.PlayerDeckList.Count-1; i++)
            {
                GameObject go = Instantiate(deckPrefab, listOfDecks.transform);
                go.GetComponent<deckButtons>().
                    SetTitle(DeckManager.Instance.PlayerDeckList[DeckManager.Instance.currentSelectedDeckNum].DeckName);
            }
        }
        else //just edited the name of a previous deck
        {
            listOfDecks.transform.GetChild(DeckManager.Instance.currentSelectedDeckNum)
                .gameObject.GetComponent<deckButtons>().SetTitle(DeckManager.
                Instance.PlayerDeckList[DeckManager.Instance.currentSelectedDeckNum].DeckName);
        }
    }
    else
    {
        print("Im inside ==0");
        foreach (var val in DeckManager.Instance.PlayerDeckList)
        {
            GameObject go = Instantiate(deckPrefab, listOfDecks.transform);
        }
    }
    if (DeckManager.Instance.PlayerDeckList.Count == 8)
    {
        addDeckBtn.GetComponent<Button>().interactable = false;
    }
}
```

Figure 27: Load Decks from Firebase

To begin with, it check's if there are already some cards inside the panel. If there are not, then that means that the panel is instantiated for the first time. If there are, then it finds the differences from the DeckManager and adds them.

4.2.2 DataBridge

DataBridge is a class that's authorized to communicate with the database, similar to the Register class from before.

```
private string DATA_URL = "[redacted]";
```

```
void Awake()
{
    if (_instance != null && _instance != this)
    {
        Destroy(this.gameObject);
    }
    else
    {
        _instance = this;
    }
}
```

Figure 28: Making DataBridge Singleton

Again a URL link is required to establish the connection to the Realtime Database. I used the Singleton design pattern to ensure that it's the only instance in the game.

```
public void LoadData()
{
    //FirebaseDatabase.GetInstance(DATA_URL).GetReferenceFromUrl(DATA_URL)
    databaseReference.GetValueAsync().ContinueWith((task =>
    {
        if (task.IsCanceled)
        {
            Debug.Log("LoadData task is canceled.");
        }
        if (task.IsFaulted)
        {
            Debug.Log("LoadData task is faulted.");
        }
        if (task.IsCompleted)
        {
            DataSnapshot snapshot = task.Result;

            string playerData = snapshot.Child("Users").Child(FirebaseAuth.DefaultInstance.CurrentUser.UserId).GetRawJsonValue();

            Player m = JsonUtility.FromJson<Player>(playerData);

            foreach (var child in snapshot.Child("Users").Children)
            {
                string t = child.GetRawJsonValue();
                Player extractedData = JsonUtility.FromJson<Player>(t);

                if (extractedData.Pid == FirebaseAuth.DefaultInstance.CurrentUser.UserId)
                {
                    print("Username of p is: " + extractedData.Username);
                    print("pid of p is: " + extractedData.Pid);

                    UIManager.Instance.SetUsername(extractedData.Username);

                    //string t2 = child.Child("Decks").GetRawJsonValue();
                    //Deck extractedData2 = JsonUtility.FromJson<Deck>(t2);


                    foreach (var deck in child.Child("Decks").Children)
                    {
                        string t2 = deck.GetRawJsonValue();
                        Deck extractedData2 = JsonUtility.FromJson<Deck>(t2);

                        DeckManager.Instance.PlayerDeckList.Add(extractedData2);
                        print("deck value is: " + extractedData2.PlayerDeck);
                    }
                }
```

Figure 29: Load decks from Firebase

Then whenever the player logins to the game the DataBridge attempts to load all of his decks from the database. It takes a snapshot of the database and I use it to extract a string in Json format. Afterwards, I do a reverse format from Json to the Player class named extractedData. I check all the Personal ID's until I find the one matching the currently logged in instance's ID. Lastly, I search for all the decks under his ID, and again I convert them from Json to a Deck class variable called extractedData2. In closing, that is how the loading functions operate.

In addition, when the player exits the deck panel, a save is performed automatically.

```
public void SaveDeckName(string deckName)
{
    string jsonData = JsonUtility.ToJson(data);
    databaseReference.Child("Users").
        Child(FirebaseAuth.DefaultInstance.CurrentUser.UserId).Child("Decks").
        Child(DeckManager.Instance.currentSelectedDeckNum.ToString()).
        Child("DeckName").SetRawJsonValueAsync(jsonData).ContinueWith((task =>
    {
        if (task.IsCanceled)
        {
            Debug.Log("SaveData task is canceled.");
        }
        if (task.IsFaulted)
        {
            Debug.Log("SaveData task is faulted.");
        }
        if (task.IsCompleted)
        {
            print("SaveData is completed");
        }
    }));
}
```

Figure 30: Save deck to Firebase

Again the process is very similar to the Registration. I create a string based on the Json format of his personal ID.

4.3 Deck Manager

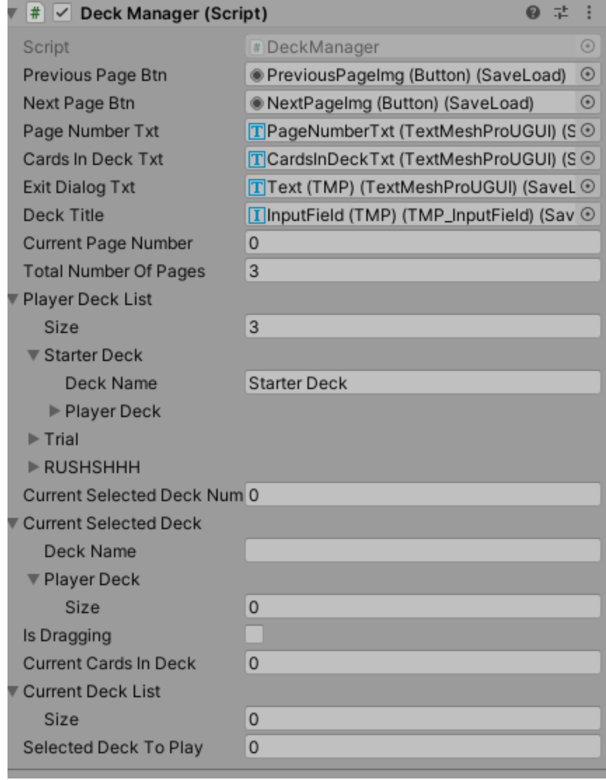

Figure 31: Deck Manager Inspector

The deck manager, gets the information from the database during the players first login. Then it has a structure similar to the database, to save all the players decks. In that way when the player requests information for his decks, I can pull this information directly from the Deck Manager which is stored locally, instead of trying to load them from the cloud. The only time the deck manager actually communicates with the cloud database, is when the user saves a new deck, or when the player loads into the main menu.

4.4 Scriptable Objects and Cards.

4.4.1 Cards Instantiation

It's vital for the game to be scalable. For that reason, I created the class "Card" which is inherited to all of our cards.

```
using UnityEngine;
[System.Serializable]

[CreateAssetMenu(fileName = "New Card", menuName = "Card")]
public class Card : ScriptableObject
{

    public int id;
    public string cardname;
    public int cost;
    public int attack;
    public int health;
    public string description;
    public Sprite artworkImage;

    public Card()
    {

    }

    public Card(int Id, string Name, int Cost, int Attack, int Health, string Description, Sprite Artwork)
    {
        id = Id;
        cardname = Name;
        cost = Cost;
        attack = Attack;
        health = Health;
        description = Description;
        artworkImage = Artwork;
    }
}
```

Figure 32: Card Class

Each card saves some values like an ID, its name, the cost, etc.

I added an option on the editor to create a new card, simply by right clicking.

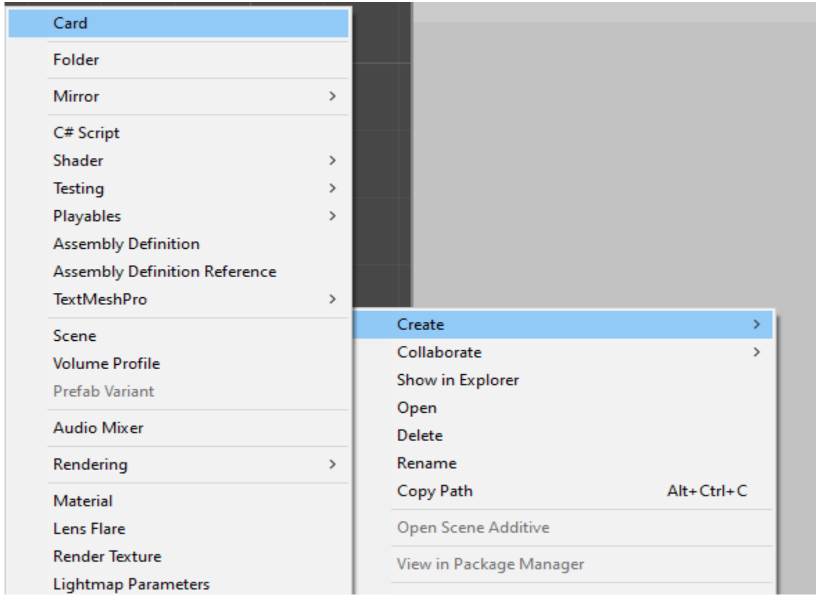

Figure 33: Creating Cards from Menu

This option creates a scriptable object that represents our card. Next, I can edit the values that we want the card to have.

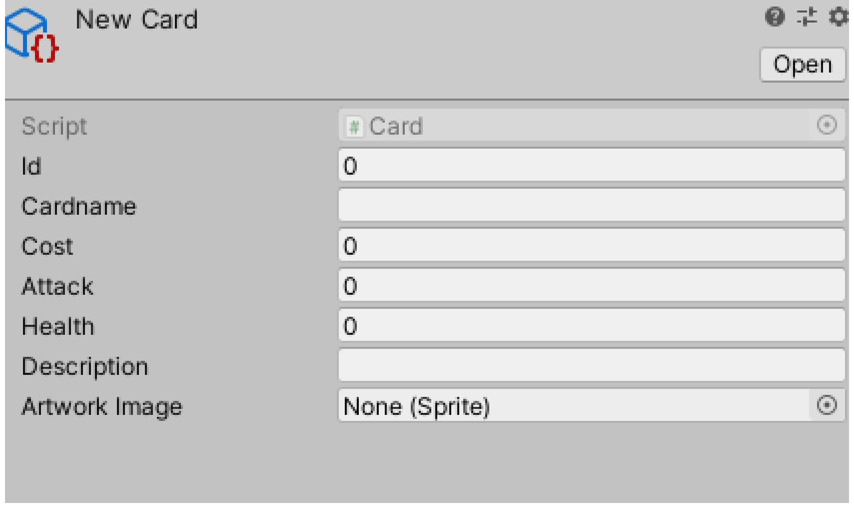

Figure 34: Scriptable Objects of type Card

Lastly I dragged the scriptable object to a card database that contains a list with all the cards I want to use.

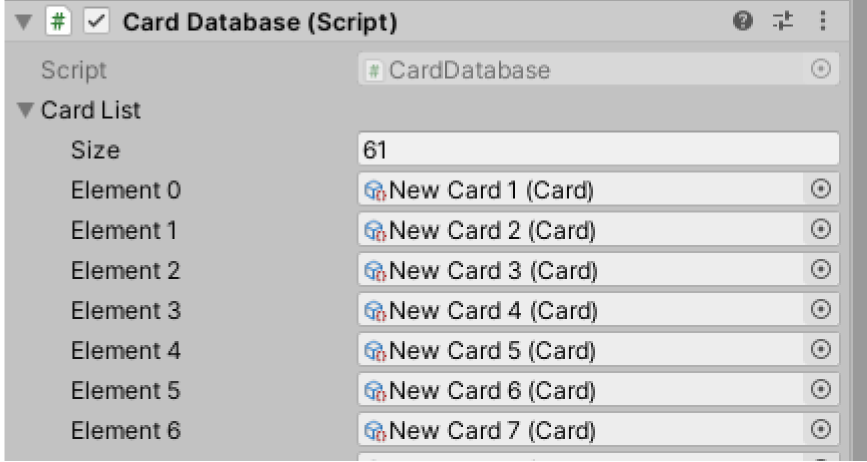

Figure 35: Card Database

Now I just instantiate a card prefab over and over again and I simply assign it a scriptable object. Then I added a CardDisplay class which is responsible to load the values from the scriptable object to the actual prefab.

```csharp
public class CardDisplay : MonoBehaviour
{
    public Card card;

    public TextMeshProUGUI nameText;
    public TextMeshProUGUI manaText;
    public TextMeshProUGUI attackText;
    public TextMeshProUGUI healthText;
    public TextMeshProUGUI descriptionText;
    public Image artworkImage;
    public GameObject cardBackImage;
    public bool isCardBackActive;



    void Start()
    {
        if (card != null)
        {
            nameText.text = card.cardname;
            manaText.text = card.cost.ToString();
            healthText.text = card.health.ToString();
            attackText.text = card.attack.ToString();
            descriptionText.text = card.description;
            artworkImage.sprite = card.artworkImage;
        }
    }
```

Figure 36: Card Display Script

4.4.2 DropZones

Every Drop zone area in the game has a DropZone script attached to is. This script contains some event handlers that trigger when the mouse pointer has entered or exited an area. When this happens, they check if we drag something (like a card) and if we do, they set their parent to the specific area.

Then as the cards exit the area, they set the card’s parent back to the previous area they were.

```
public void OnPointerEnter(PointerEventData eventData)
{
    if (eventData.pointerDrag == null)
        return;
    Draggable d = eventData.pointerDrag.GetComponent<Draggable>();
    if (d != null)
    {
        //Debug.Log("ENTER this.tag = " + this.tag);
        d.placeholderParent = this.transform;
    }
}

public void OnPointerExit(PointerEventData eventData)
{
    if (eventData.pointerDrag == null)
        return;
    Draggable d = eventData.pointerDrag.GetComponent<Draggable>();
    if (d != null && d.placeholderParent == this.transform)
    {
        //Debug.Log("EXIT this.tag = " + this.tag);
        d.placeholderParent = d.parentToReturnTo;
    }
}
```

Figure 37: Dropzones Script

If however, the player drops the card inside the area, then they set the previous parent to the zone. (their current parent already changed when they entered the area, with the OnPointerEnter method)

```
public void OnDrop(PointerEventData eventData)
{
    //Debug.Log(eventData.pointerDrag.name + " was dropped on "+gameObject.name);

    Draggable d = eventData.pointerDrag.GetComponent<Draggable>();
    if (d != null)
    {
        if(d.placeholderParent != (enemyHand.transform) || d.placeholderParent != (enemytabletop.transform))
        {
            d.parentToReturnTo = this.transform;
        }
        else
        {
            //Debug.Log("alliws gurna pisw xwris na kaneis tpt");
        }
    }
}
```

Figure 38: Dropzones Script part 2

4.4.3 Card movement

Every card has a “draggable” script which allows the player to move the card. Inside the script there are drag handlers that trigger when the user begins to move a card, when it moves and when the movement ends.

The OnBeginDrag takes as a parameter the pointer event that gets generated when the player clicks on a card. We get the network identity of the player (so we know who tries to move the card) and we store it. After that we instantiate the placeholder which is a copy of the card, and shows the player where his card is going to be placed.

I also store the container of this card as parentToReturnTo, in-case an invalid move takes place, so I will know where to return the card to. And I also save the current parent in placeholderParent.

Last but not least, while I move the card around, I want my mouse to be able to RayCast what's bellow the card. However, my card blocks any ray cast from the mouse cursor; So I temporarily set the blocksRaycasts to false.

```
public void OnBeginDrag(PointerEventData eventData)
{
    if (!isDraggable) return;

    NetworkIdentity networkIdentity = NetworkClient.connection.identity;
    PlayerManager = networkIdentity.GetComponent<PlayerManager>();

    placeholder = Instantiate(PlayerManager.cardPrefab);
    placeholder.GetComponent<CardDisplay>().card = this.gameObject.GetComponent<CardDisplay>().card;/
    placeholder.transform.SetParent(this.transform.parent, false);
    placeholder.GetComponent<CanvasGroup>().alpha = 0.27f;
    placeholder.GetComponent<CanvasGroup>().blocksRaycasts = false;
    LayoutElement le = placeholder.AddComponent<LayoutElement>();
    le.preferredHeight = this.GetComponent<LayoutElement>().preferredHeight;
    le.preferredWidth = this.GetComponent<LayoutElement>().preferredWidth;
    le.flexibleHeight = 0;
    le.flexibleHeight = 0;

    placeholder.transform.SetSiblingIndex(this.transform.GetSiblingIndex ());

    //Debug.Log("placeholder name in draggable is " + placeholder.name);
    parentToReturnTo = this.transform.parent;
    placeholderParent = parentToReturnTo;
    this.transform.SetParent(this.transform.parent.parent);

    GetComponent<CanvasGroup>().blocksRaycasts = false;
    //isDragging = true;
}
```

Figure 39: Draggable Script

When the player finally starts to drag the card around the OnDrag method activates.

First, we need to check if the card is draggable and if it is the players turn to play.

There are some conditions in place that restrain the players from doing illegal moves such as dropping the cards on the opponent's area etc. If all the conditions are true, then I set the placeholder parent to be equal to the latest placeholderParent value, which was changed within our DropZone script.

Lastly, I keep track of the position of the card in the area it starts, so if it returns to the previous parent it will be placed in the same spot. I do that, by comparing the x value of the placeholder with the x value of the other cards and determining its sibling index.

```
public void OnDrag(PointerEventData eventData)
{
    if (!isDraggable) return;
    //Debug.Log("OnDrag, isDraggable is " + isDraggable);
    // Debug.Log("OnDrag");
    this.transform.position = eventData.position;

    if (!PlayerManager.isMyTurn)
    {
        tabletop.transform.GetComponent<Image>().raycastTarget = false;
    }
    else
    {
        tabletop.transform.GetComponent<Image>().raycastTarget = true;
    }

    if (gameObject.transform.parent!= placeholderParent && placeholderParent
        != enemyHand.transform && placeholderParent != enemytabletop.transform && PlayerManager.isMyTurn)
    {
        placeholder.transform.SetParent(placeholderParent);
    }

    newSiblingIndex = placeholderParent.childCount;
    for (int i = 0; i < placeholderParent.childCount; i++)
    {
        if (this.transform.position.x < placeholderParent.GetChild(i).position.x)
        {
            newSiblingIndex = i;
            if (placeholder.transform.GetSiblingIndex() < newSiblingIndex)
            {
                newSiblingIndex--;
            }
            break;
        }
    }
    placeholder.transform.SetSiblingIndex(newSiblingIndex);
}
```

Figure 40: Draggable Script part 2

The OnEndDrag method is called when the player lets go of the card.

Firstly, I check if the card is played in the proper area, if the player has the required mana, and if the board is not full. Then if all these requirements are true, I subtract the cost from the players mana pool and I change the parent to the board. Then with the command “PlayerManager.PlayCard” I send a command to the server that I played a card. With the command I also send some parameters as to the identity of the card that is played and where I played it too. If the board is full, then I place the card back to the hand.

It should be noted that the player can change the card order of his hand anytime he wishes. That is why I also check if the destination area is the hand. If it is, I let him re-order the cards as he wishes.

Lastly, I set the blockRaycast back to true and I destroy the placeholder.

```
public void OnEndDrag(PointerEventData eventData)
{
    if (!isDraggable) return;
    //isDragging = false;

    //PlayerManager.CmdSpawnPreview(gameObject.GetComponent<CardDisplay>().card.id, placeholderParent);
    if (placeholder.transform.parent == tabletop.transform && GameManager.Instance.currentMana >= gameObject.GetComponent<CardDisplay>().card.cost)
    { //here i want to play the card
        if(tabletop.transform.childCount < GameManager.Instance.maxCardsOnBoard) //if board ISNT full then play it
        {
            isDraggable = false;
            GameManager.Instance.currentMana -= gameObject.GetComponent<CardDisplay>().card.cost;
            this.transform.SetParent(parentToReturnTo);
            this.transform.SetSiblingIndex(placeholder.transform.GetSiblingIndex());
            PlayerManager.PlayCard(gameObject, placeholder.transform.parent, newSiblingIndex);
        }
        else //if board IS full, return it to my hand
        {
            this.transform.SetParent(hand.transform);
            this.transform.SetSiblingIndex(placeholder.transform.GetSiblingIndex());
            Debug.Log("not enough space on board");
        }
    }
    else if (placeholder.transform.parent == hand.transform)
    { //here i move the card in my hand
        this.transform.SetParent(parentToReturnTo);
        this.transform.SetSiblingIndex(placeholder.transform.GetSiblingIndex());
        PlayerManager.CmdPlayCard(gameObject, placeholder.transform.parent, newSiblingIndex);
    }
    else //here i return the card to my hand (THOUGHT: possibly because out of mana?)
    {
        this.transform.SetParent(hand.transform);
        this.transform.SetSiblingIndex(placeholder.transform.GetSiblingIndex());
        Debug.Log("not enough mana");
    }

    GetComponent<CanvasGroup>().blocksRaycasts = true;
    Destroy(placeholder);
    //Debug.Log("end of drag");
```

Figure 41: Draggable Script Part 3

4.4 GameManager

The GameManager is a class responsible for the main functionality in the game. The GameManager acts like the "dealer" in a game of cards. Basically, it takes care of providing the players with cards, managing their resources like mana, keeping track of what phase the game is in, and changing the turns between the players. One of the main things that the GameManager does, is the initialization of some classes.

```
private void Start()
{
    turnText = GameObject.FindWithTag("TurnText").GetComponent<TextMeshProUGUI>();
    manaText = GameObject.FindWithTag("ManaCrystal").GetComponentInChildren<TextMeshProUGUI>();
    playerDeck = GameObject.FindWithTag("PlayerDeck").GetComponent<PlayerDeck>();
    hand = GameObject.FindWithTag("Hand");
    tabletop = GameObject.FindWithTag("Tabletop");
    mulliganPanel = GameObject.FindWithTag("MulliganPanel");
    playerPortrait = GameObject.FindWithTag("Player");
    enemyPortrait = GameObject.FindWithTag("EnemyPlayer");

    currentBattlePhase = BattlePhase.None;
    minionSelected = null;
    Cursor.SetCursor(defaultCursor, Vector2.zero, CursorMode.ForceSoftware);
    endTurnButton.interactable = false;
    startButton = mulliganPanel.GetComponent<MulliganPanel>().GetStartGameButton();
    mulliganButton = mulliganPanel.GetComponent<MulliganPanel>().GetMulliganButton();
    keepButton = mulliganPanel.GetComponent<MulliganPanel>().GetKeepButton();
    waitingPlayerText = mulliganPanel.GetComponent<MulliganPanel>().GetPlayerText();
}
```

Figure 42: GameManager Script

Most of the things in our scene, like the texts and the drop zones, are spawned dynamically when the players connect to each other. For that reason, I cannot simply assign the variables from the editor. So, I assigned every important thing with the appropriate tag and used the FindWithTag command to assign them correctly. After that I initialized some of the variables to their first value.

I used two Enum variables, one that keeps track of the phase each player is and the second keeps track of what battle phase the attacking player is in.

```
enum GameState { FlipCoin, Mulligan, PlayerTurn, EndGame };
GameState currentGameState;

enum BattlePhase { None, Selected, Targeted}
BattlePhase currentBattlePhase;
```

Figure 43: Enums

In the ChangeGameState method, I pass as a parameter the gamestate Enum. Then, an “if” argument checks the stage of the game.

If the players are in the mulligan state then they draw 5 cards each from their decks.

If the players are in the PlayerTurn stage, a nested “if” checks if it’s the players turn. If it is, we set the canAttack of our cards that we have played to true. We also enable the green flames around them to indicate it to the player. Finally we draw a card.

If it’s not our Turn we set the canAttack of our cards to false and also we disable the green flames by changing the alpha channel of the color to 0.

```
public void ChangeGameState(GameState gameState)
{
    if (gameState == GameState.FlipCoin)
    {

        currentGameState = GameState.FlipCoin;
    }
    else if (gameState == GameState.Mulligan)
    {
        currentGameState = GameState.Mulligan;
        for (int i = 0; i < 5; i++)
        {
            playerDeck.Draw();
        }
    }
    else if (gameState == GameState.PlayerTurn)
    {
        NetworkIdentity networkIdentity = NetworkClient.connection.identity;
        playerManager = networkIdentity.GetComponent<PlayerManager>();

        if (playerManager.isMyTurn)//here do things when my turn or enemies turn starts
        {
            endTurnButton.interactable = true;
            for (int i = tabletop.transform.childCount - 1; i >= 0; --i)//Set all played minions canAttack to true
            {
                Transform child = tabletop.transform.GetChild(i);
                if (child.gameObject.tag == "Card" && !child.gameObject.GetComponent<Draggable>().canAttack)
                {
                    child.gameObject.GetComponent<Draggable>().canAttack = true;
                    child.gameObject.GetComponent<Image>().material = greenFlame;
                }
            }
            playerDeck.Draw();
        }
        else
        {
            endTurnButton.interactable = false;
            for (int i = tabletop.transform.childCount - 1; i >= 0; --i)//Set all played minions canAttack to false
            {
                Transform child = tabletop.transform.GetChild(i);
                if (child.gameObject.tag == "Card")
                {
                    child.gameObject.GetComponent<Draggable>().canAttack = false;
                    Color c = child.gameObject.GetComponent<Image>().color;
                    c.a = 0;
                    child.gameObject.GetComponent<Image>().color = c;
                }
            }
        }
        currentGameState = GameState.PlayerTurn;
    }
```

Figure 44: GameManager script part 2

I also manage the end game messages from the GameManager.

```
public void WonGame()
{
    endGamePanel.SetActive(true);
    endGamePanel.GetComponentInChildren<TextMeshProUGUI>().SetText("Congratulations, You Won!");
}
public void LostGame()
{
    endGamePanel.SetActive(true);
    endGamePanel.GetComponentInChildren<TextMeshProUGUI>().SetText("Unfortunately, You Lost!");
}
public void RestartGame()
{
    SceneManager.LoadScene(1);
}
```

Figure 45: End Game Clauses

If the player wins or loses, I display the appropriate message to the screen. I also have a RestartGame method that loads the main menu scene when the player clicks the Restart button.

4.5 Mirror implementation

Mirror needs a server in order for the players to be able to connect and play together. However, in Host mode a player can act both as a client and a server. The drawback of this method is that a client has to connect to another client and sometimes that is difficult due to security reasons like firewalls. So, the client that acts as the host must port-forward their rooter and allow incoming connections to their computer. A dedicated server would solve this problem, but it requires a monthly fee, and in the scope of this thesis I would like to keep the project free.

4.5.1 Authority

In Mirror, the term "authority" describes the process of determining who owns and controls an object. Authority is divided in two parts. Server authority and Client authority.

The term “server authority” means that the server has control of an object. By default, the server has authority over an object. All collectible things, moving platforms, NPCs, and other networked objects that aren't the players would be managed and controlled by the server.

Similarly, “client authority” means that the client has control of the object.

When a client has authority over an object, they can issue commands and the object will be destroyed automatically when the client disconnects.

SyncVars and other serialization features are still controlled by the server, even if a client has authority over an object. For a component to sync with other clients, it must utilize a Command to update the server's state.[9]

4.5.2 Attributes

To make NetworkBehaviour scripts run on either the client or the server, networking attributes are added to member functions.

These attributes can be utilized in Unity game loop methods like Start and Update, as well as additional methods that have been implemented.[10]

[Server] and [Client] attributes means that the following method can only be called by the server or a client respectively.

[Command] is called from a client to run on the server. For example, when a client wants to deal some damage on the opponents cards, he sends a command to the server with the damage he deals and then the server takes the damage value and applies it to the other client.

[TargetRPC] and [ClientRPC] indicate that the method uses a Remote Procedure Call (RPC).

RPC’s can only be called by the server.

4.5.3 Remote Procedure Calls

RPCs are split in two parts, ClientRPC and TargetRPC. Their functionality is the same, but to whom they address is not.

ClientRpc calls are sent from server objects to client objects. There are no security issues with server objects being able to send these calls because the server has authority. Basically, when server takes an action and wants all the clients to know about it then an RPC is called. For example, if a player's life points change and we want every player that is connected to receive this change, we can do a ClientRPC call and pass the new life points.

TargetRPC is similar but it can only target one specific client. We call this when we do not want every client to know a change.

Command and ClientRpc parameters are serialized and transferred across the network.[11]

4.5.4 PlayerManager

The script that is responsible for the communications between the players is the PlayerManager.

```
public bool isMyTurn = false;

[SyncVar(hook = nameof(SetPlayersReady))]
public int playersReady;

public override void OnStartClient()
{
    base.OnStartClient();

    hand = GameObject.FindWithTag("Hand");
    tabletop = GameObject.FindWithTag("Tabletop");
    enemyHand = GameObject.FindWithTag("EnemyHand");
    enemytabletop = GameObject.FindWithTag("EnemyTabletop");
    mulliganPanel = GameObject.FindWithTag("MulliganPanel");

    if (isClientOnly)
    {
        isMyTurn = true;
        CmdChangeTurn();
    }
}
```

Figure 46: PlayerManager Script

As soon a client connects to the game, I set his turn to false. Then I have a variable that is synched across both clients. I initialize in real time the drop zones (as discussed earlier). Then if the player is only the client and not both the client and the server I change his turn true. Finally, I call a command to change the turn for the client, network side.

```
public void SetPlayersReady(int oldPlayers, int newPlayers)
{
    GameManager.Instance.playersReady = newPlayers;
    if(newPlayers == 2)
    {
        GameManager.Instance.ChangeGameString("Mulligan");
    }
}
```

Figure 47: Initialize players turn

SetPlayersReady is a method that checks if both players are ready. If they are, it changes both their game state to the Mulligan phase, implying that the game has started.

```
public void DealCards(int id)
{
    if (GameManager.Instance.currentGameState == GameManager.GameState.Mulligan)
    {
        CmdMulliganCards(id);
    }
    else if (GameManager.Instance.currentGameState == GameManager.GameState.PlayerTurn)
    {
        CmdDealCards(id);
    }
    else
    {
        Debug.Log("CmdDealCards Unknown GameState");
    }
}
```

Figure 48: DealCards script

Every time a player needs to get some cards, DealCards is called. It has an integer parameter called ID which is the ID of the card he is going to get. Then I check if the game is in the Mulligan phase or if it’s on the normal turn phase. According to the phase a different command will be called.

```
[Command]
public void CmdMulliganCards(int id)
{
    foreach (Card card in CardDatabase.Instance.cardList)
    {
        if (card.id == id)
        {
            GameObject go;
            go = Instantiate(cardPrefab);
            NetworkServer.Spawn(go, connectionToClient);
            go.GetComponent<CardDisplay>().card = card;

            //Debug.Log("I've drawn a " + card);
            RpcShowCard(go, id, "Mulligan");
        }
    }
}
```

Figure 49: Command for Mulligan

The command CmdMulliganCards is called when it's the mulligan phase and the player has to draw a card. I take the ID of the card and I search the card database list which has all the available cards, in order to instantiate it. After I instantiate the card, I set the CardDisplay to attach the proper values to the card and then I call the RpcShowCard which is responsible to show the card to each player correctly.

```
[ClientRpc]
void RpcShowCard(GameObject go,int id, string Type)
{
    if (Type == "Dealt")...
    else if (Type == "Mulligan")
    {
        if (hasAuthority)
        {
            Debug.Log("RPCshowcard Mulligan HAS authority");
            foreach (Card card in CardDatabase.Instance.cardList)
            {
                if (card.id == id)
                {
                    go.GetComponent<CardDisplay>().card = card; //does it really do something here?
                }
            }
            go.transform.SetParent(mulliganPanel.transform, true);
            go.GetComponent<CanvasGroup>().blocksRaycasts = false;
        }
        else
        {
            Debug.Log("RPCshowcard Mulligan NO authority Destroy");
            //Destroy(go);
            foreach (Card card in CardDatabase.Instance.cardList)
            {
                if (card.id == id)
                {
                    go.GetComponent<CardDisplay>().card = card; //does it really do something here?
                }
            }
            go.transform.SetParent(enemyHand.transform, false);
            go.GetComponent<CanvasGroup>().blocksRaycasts = false;
            go.GetComponent<CardDisplay>().FlipCard();
        }
    }
}
```

Figure 50: Rpc to show cards in Mulligan

RpcShowCard is a call that goes to all available clients as it has the ClientRpc tag. Here I check if the type of the dealt card from earlier is Mulligan, and if it is, I check if the current player has authority over the card. If he has, then I spawn the card on the mulligan panel which is the place I want the mulligan cards to go. If the player doesn't have authority over the card then it means that the card is not the players therefore it has to be his opponent's. In that case, I spawn the card in the opponent's hand.

```
[ClientRpc]
void RpcShowCard(GameObject go,int id, string Type)
{
    if (Type == "Dealt")
    {
        if (hasAuthority)
        {
            foreach (Card card in CardDatabase.Instance.cardList)
            {
                if (card.id == id)
                {
                    go.GetComponent<CardDisplay>().card = card;

                }
            }
            if (hand.transform.childCount < GameManager.Instance.maxCardsInHand)
            {
                go.transform.SetParent(hand.transform, false);
            }
            else
            {//Here i can burn the card, because my hand is full
                Debug.Log("HAND IS FULL!");
                Destroy(go);
            }

        }
        else
        {
```

Figure 51: Rpc to show cards in hand

The same logic goes with all the cards a player can draw. I check the authority of the player on the current card and I spawn it in his hand if he has it, or at his opponent's hand if he doesn't. One more check that I do before a player draws a card, is to see if he has less or more cards than the maximum amount allowed.

```
[ClientRpc]
void RpcPlayCard(GameObject card, Transform placeholderParent, int index)
{
    if (hasAuthority)
    {
        card.transform.SetParent(placeholderParent);
        card.transform.SetSiblingIndex(index);
    }
    else
    {
        if(placeholderParent == hand.transform)
        {
            Debug.Log("Eimai sto RPCplaycard, NO authority, enemyhand");
            card.transform.SetParent(enemyHand.transform, false);
            card.transform.SetSiblingIndex(index);
        }
        else if(placeholderParent == tabletop.transform)
        {
            Debug.Log("Eimai sto RPCplaycard, NO authority, enemytabletop");
            card.GetComponent<CanvasGroup>().blocksRaycasts = true;
            card.transform.SetParent(enemytabletop.transform, false);
            card.transform.Rotate(0f, 0f, 180f);
            card.transform.SetSiblingIndex(index);
            card.GetComponent<CardDisplay>().FlipCard();
        }
    }
}
```

Figure 52: Rpc to play card

In RpcPlayCard, a player has given the command to the server to play a card. So the server checks if it's the player's card. If it is it just plays it. If it is not, then the enemy player either moved the position of the card at his hand, or he played it on the table top dropzone. If he played it on the table top I make sure I disable the raycasts on the card and I make sure to rotate it, in order to look towards the player. Finally, I flip the card, which means both players can see what card was played.

```
[ClientRpc]
void RpcChangeTurn()
{
    PlayerManager pm = NetworkClient.connection.identity.GetComponent<PlayerManager>();
    pm.isMyTurn = !(pm.isMyTurn);

    //GameManager.Instance.currentGameState = GameManager.GameState.PlayerTurn;
    GameManager.Instance.ChangeGameState(GameManager.GameState.PlayerTurn);
    if (GameManager.Instance.maxMana < 10 && pm.isMyTurn)
        GameManager.Instance.maxMana++;

    GameManager.Instance.currentMana = GameManager.Instance.maxMana;
    GameManager.Instance.ReloadText();
}
```

Figure 53: Rpc change turn

In RpcChangeTurn, I change the players turn. In order to do it, I just flip the values between each player manager. The client only starts with his turn set to true and the host starts with his turn set to false. In that way, I simply switch turns back and forth. Also, many of the things that happen on the start of each round, happen here. For example, it increases the player's maximum mana by 1 and it also refills it to the maximum value.

5. Network Infrastructure – Firebase Services

As I wrote previously in Chapter 2, Firebase offers an array of services, some of them which I used in this project. More specifically I used Firebase's Authentication and Realtime Database services.

5.1 Firebase Authentication

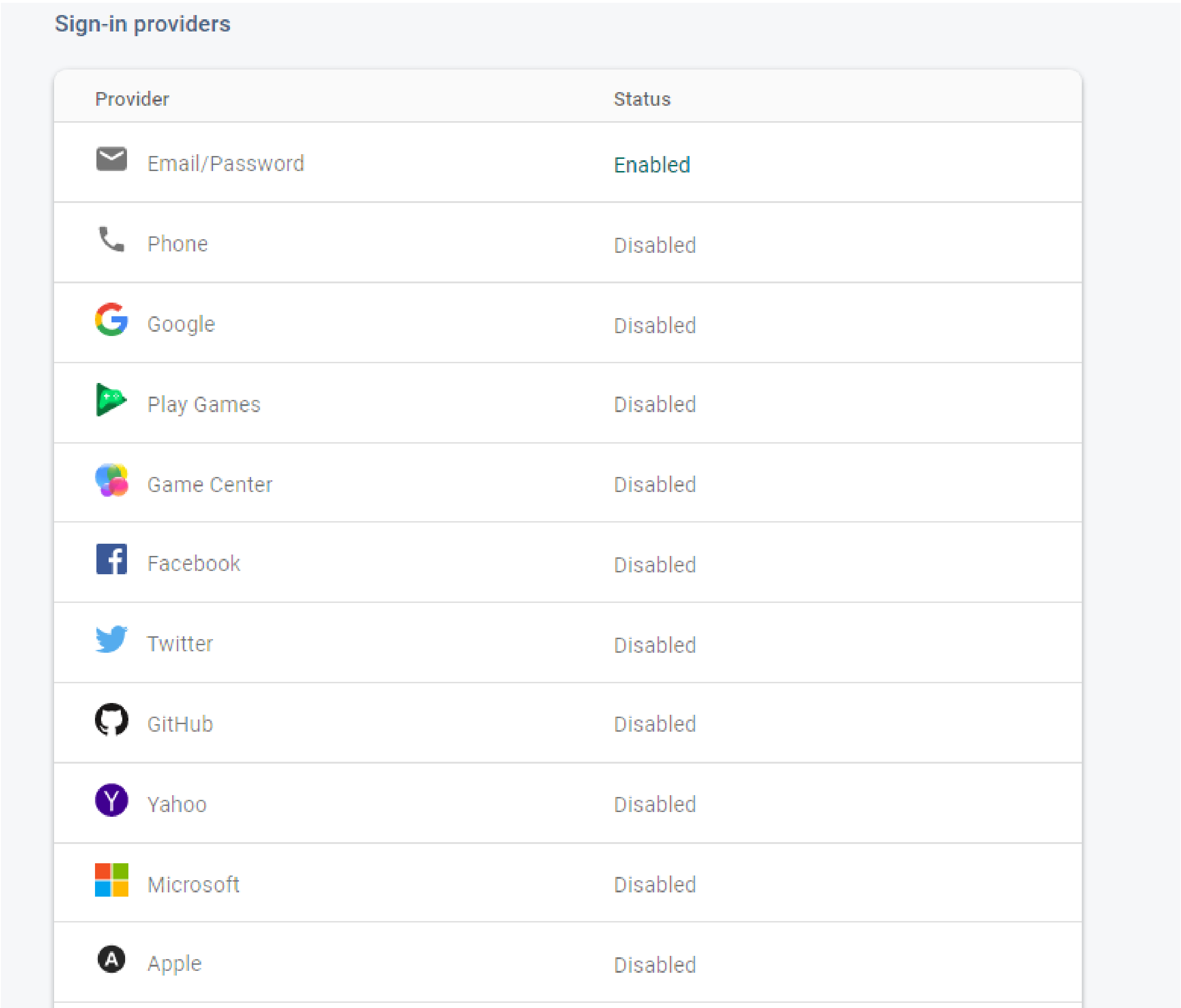

Figure 54: Firebase Authentication Sign-in methods

With Firebase Authentication we can use a lot of different platforms to login. It is all integrated and we don't have to do a lot of changes to implement the sign-in with other platforms. However, for the purpose of this thesis I have only enabled to be able to login with an email-password combination which the user has to set in the game.

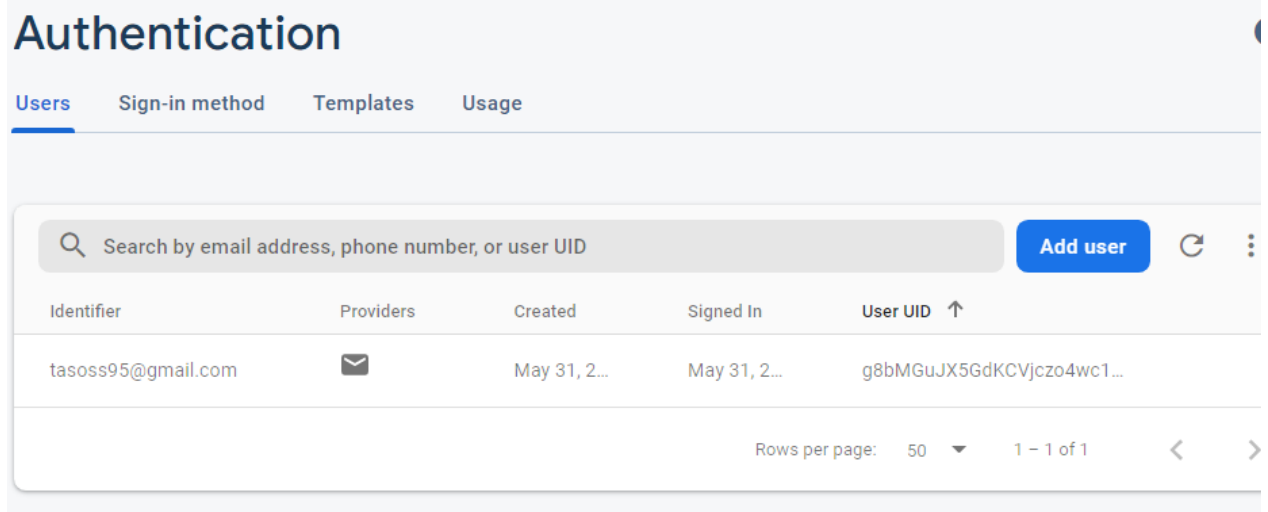

Figure 55: Authentication in Firebase

When we register a user, firebase ties his email address and his password to a unique User ID.

5.2 Realtime Database

As soon as the registration of a user happens, I create under the child Users a child named after the unique ID from the Authentication process. That way we can make sure that each user has its own "space" in the database under his unique ID. Another benefit of this, is that we can allow multiple players to have the same name if they wish, as we do not use it to verify each player.

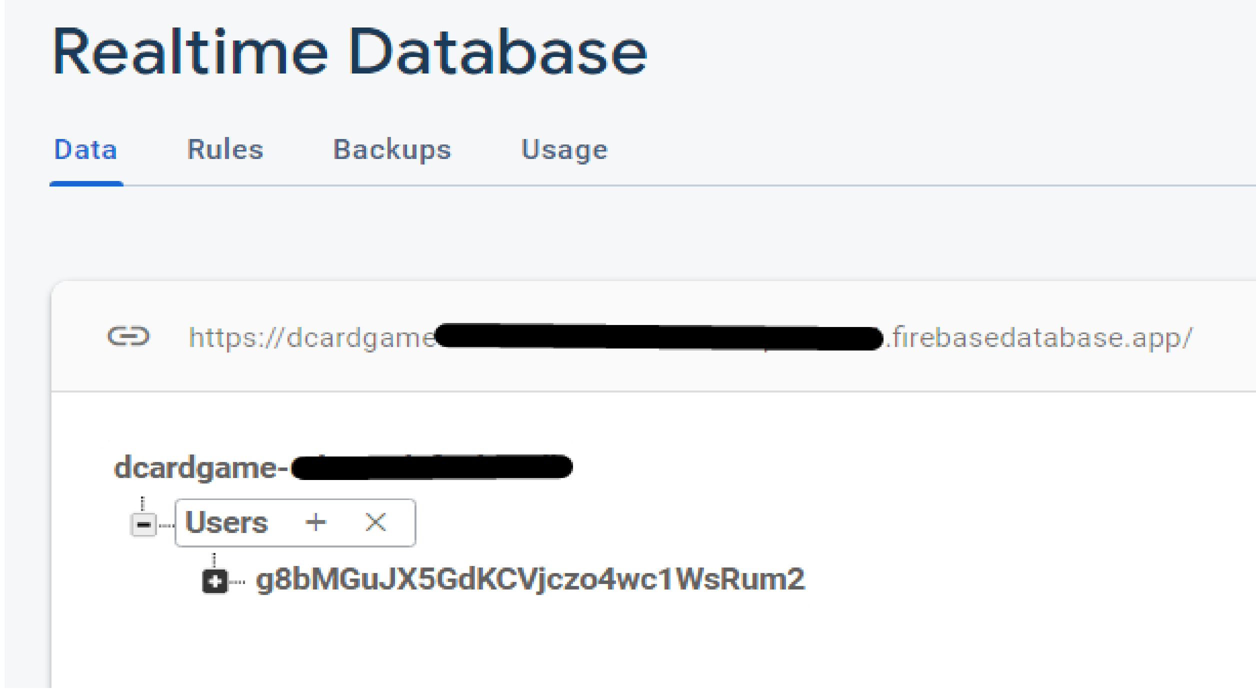

Figure 56: Realtime Database

As we can see in the example above, the User ID is the same as the User ID from the authentication process. This means that it is the same user.

5.2.1 The structure of the Realtime Database

If we open the UID we can observe the format that we store information.

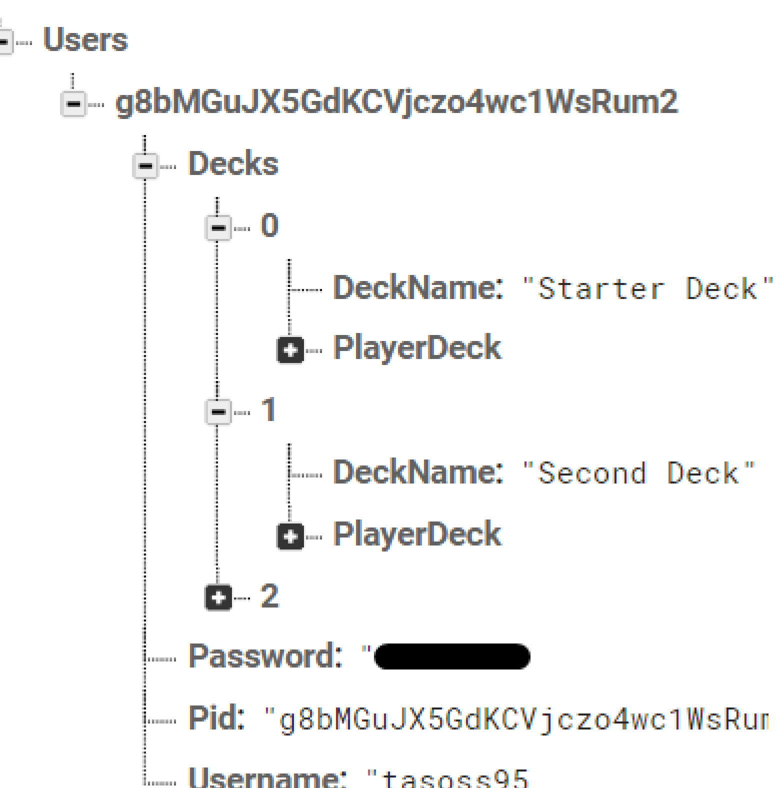

Figure 57: RealTime Database structure

Under each UID, we have 4 children. Decks, Password, PID, and username.

The password field is used just for debugging purposes. It's not encrypted in any way and it will be removed when the game reaches the production stage, as it is considered a security hole.

In addition, I store the PID, which can be used as queries inside the game. Such queries can be used when we are looking for a specific user to save or load his decks.

In the next field I store the Username of the user. I can get the username from here and display it in-game.

And last but not least, I have the Decks child, where I save each players deck.

Inside the Decks I have an index for each deck, starting from 0. If we open the index, we can find each decks name and another child named PlayerDeck.

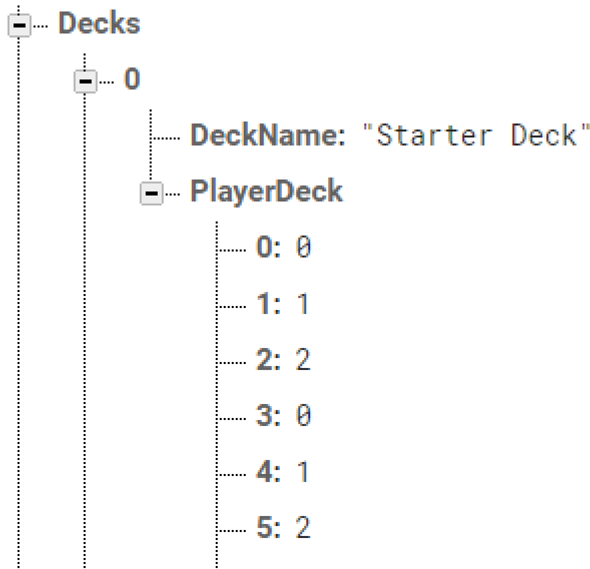

Figure 58: Cards ID's in Database

PlayerDeck simply stores all the card ID’s that the specific deck has.

6. Conclusion

6.1 Problems faced during development

During development, I’ve faced a number of problems and set backs. Most of them were mainly from the lack of experience in creating games. This thesis concludes my first big project with Unity. Thankfully, there are a lot of resources to seek help from, such as many YouTube Tutorials[12] and sites like Stack Overflow. Also, the community of Unity is huge and helpful, trying to assist new developers, that join their favorite game engine, with whatever problem they might face.

Another big issue that I had, was with the multiplayer part of the game. Surely, plugins like Mirror help tremendously, as you do not have to re-invent the wheel and brings the coding to a higher level of programming. Not having to deal with different transport protocols was lifesaving, as it would be too daunting to even attempt to create something that requires a multiplayer connection without it. Albeit all the tools I was provided, by Mirror, it still was a great effort to learn how to use them properly and efficiently. As players interact with each other, and not with some scripted NPC, I had to change and rethink all the logic behind my decisions, generally bringing the difficulty of the project to a whole new scope.

In the subject of the networking, another thing I had to think about, was how am I going to pass the application connection through the Windows and router firewalls. A temporary solution that I came up with, was to open a port on the router, in which the clients could connect too. However, this is both impractical and unsafe. Impractical because each time 2 players wanted to play with each other one of them had to port-forward a port on his router. This also meant that it would be impossible to build on mobile devices because opening a port through mobiles devices is considerably harder to do. It is also dangerous because a malicious user could exploit the open port to his own benefit.

Lastly, another problem that I faced was testing and debugging. As I had to connect two clients together, I had to “build” every time I wanted to try something in the game, which took around 5 minutes. Thankfully, I discovered another addon named ParrelSynch[13], which simply allowed me to open two editor windows and considerably reduce the testing/debugging time from 5 minutes to 10 seconds.

6.2 Improvements for the future

A lot of progress has been made with this game. However, there is still a lot to do in the future in order for the game to reach a production state. One improvement I would surely do, would be to replace the whole host-client architecture that I used, and setup a dedicated server with a matchmaking system that automatically connects you to another player. Amazon’s AWS would be a very cheap and quality option, as many AAA games use it already.

Another big improvement I would like to tackle, would be some cheat prevention mechanics. Basically, I would allow the server to verify all the time, the cards and decks that each player holds, as well as the damage they do etc. That way, a client would not be able to just change his local game using a CheatEngine.

6.3 Personal fulfillment

In conclusion, the whole process of creating your own game is a satisfying, full of emotions experience which I thoroughly enjoyed. I gained a lot of skills in the game developing area and I grew my perspective about game development in general. This whole project was enlightening to me, and it was a challenge that I can finally say I accomplished successfully.

The journey in game development, however, does not end here for me. On the contrary, it has just begun. It will be a road full of setbacks and challenges, but it will also be a road full of emotions, self-fulfillment, and personal success.

Bibliography and Sources

1. https://hearthstone.fandom.com/wiki/Hearthstone
2. https://en.wikipedia.org/wiki/Game_engine
3. https://en.wikipedia.org/wiki/Unity_(game_engine)
4. https://mirror-networking.gitbook.io/docs/general
5. https://en.wikipedia.org/wiki/Database
6. https://en.wikipedia.org/wiki/Cloud_database
7. https://medium.com/firebase-developers/what-is-firebase-the-complete-story-abridged-bcc730c5f2c0
8. https://assetstore.unity.com/packages/2d/textures-materials/tcg-cards-pack-63019
9. https://mirror-networking.gitbook.io/docs/guides/authority
10. https://mirror-networking.gitbook.io/docs/guides/attributes
11. https://mirror-networking.gitbook.io/docs/guides/communications/remote-actions
12. https://www.youtube.com/playlist?list=PLCbP9KGntfcFTL19eDZsWSkVMfXANF7-U
13. https://github.com/VeriorPies/ParrelSync