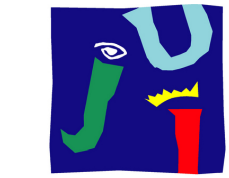

Developing a distributed, customizable multiplayer card game

Jose Angel Buforn Tena

Final Degree Work

Bachelor's Degree in
Video Game Design and Development

Universitat Jaume I

July 5, 2020

Supervised by: Manuel Francisco Dolz Zaragozá

To my parents who always had care of me.

Acknowledgments

First of all, I would like to thank my Final Degree Work supervisor, Manuel Francisco Dolz Zaragozá, PhD, for his help and being there whenever I needed, even when outside his working hours.

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

To all the people in Unity and Bolt forums trying to help with all the things where I got stuck.

Also, the ones in the Bolt's discord server [10], including some of the authors of the multiplayer system whose bugs reported were fixed or given workarounds for this project.

I would like to thank all my classmates, this was the best class from all I was, everybody helping others and no signs of belligerence in all the years. That was amazing.

I would also thank all my family for supporting me from the distance, these has been hard days.

Abstract

This document is the technical proposal of the Bachelor's Degree in Video Game Design and Development Thesis. In the thesis, will be developed a video game that emulates a real card game in most of its aspects, including different screens for each of the places where the cards should be.

The application stands out for the interaction between them wirelessly in real time emulating a realistic interaction between the players.

Contents

CHAPTER 1

Introduction

Contents

In this chapter we provide the motivation and the initial conditions taken to carry out this project. Also, we enumerate some subjects of the Video Game Design and Development degree which have been coursed and that are closely related to the project topic.

1.1 Work Motivation

In the degree there are a lot of topics that have been covered, but there was an important gap to fill: online multiplayer games. This project is intended to cover these gaps.

Also in the vast market of card games there cannot be found any online custom card game, and almost all of them are role playing games, with predefined card positions and with only one game supported. On the other hand, there are games where the players can do anything and create their own games, but they are all offline.

With these objectives in mind this final degree work lies in to make a custom card game where the players can see the same features, with the card movement synced in real time, like if it was a real game.

1.2 Objectives

The objectives fixed for this multiplayer card game are:

- To use and learn development techniques and game programming on connected devices.
- Multiplayer support.
- To get two or more devices (PC, tablet, smartphone) communicate with each other for the completion of the game.
- To create a new original way to play by using multiple synchronized screens to enrich the game experience.

1.3 Environment and Initial State

The hardware used for this project is a computer with Microsoft Windows 7 and an internet connection. We will also test the game with an Android device for having different devices to test it.

All the work is developed with the universal game engine Unity. As the internal multiplayer API (UNET) is deprecated [8], the Photon Bolt [5] plugin will be used instead.

The Photon Bolt is a framework for developing multiplayer games. It is prepared to make games where there is an avatar controlled by the player [6], so it is needed to learn it in depth and to adapt the mechanics to make a project like this.

For the art aspect, the program used to make and modify all the images is Gimp [11].

The project will be publicly available on Github [1] with a MIT License. The MIT License can be described as: "A short and simple permissive license with conditions only requiring preservation of copyright and license notices. Licensed works, modifications, and larger works may be distributed under different terms and without source code".

1.4 Related subjects

This project covers some of the topics that have been learnt in the following list of subjectes from the Video Game Design and Development degree:

- VJ1203 - Programming I (Computing)
- VJ1204 - Artistic Expression
- VJ1208 - Programming II (Computing)

- VJ1215 - Algorithms and Data Structures
- VJ1227 - Game Engines
- VJ1228 - Multiplayer Systems and Networks
- VJ1229 - Mobile Device Applications
- VJ1236 - Sound Production and Realization Techniques

CHAPTER 2

Planning and resources evaluation

Contents

In this chapter, we provide the initial planning of the project, i.e., before the start of any project task along with the resources for starting this project. With this in mind, we also perform an estimation of the project costs.

2.1 Planning

The estimating time of all the tasks and subtasks is made based on the projects done during the degree. For the project planning, the tasks have been distributed in five big parts:

- Designing the application(s).
- Creating the art involved in the game.
- Creating the audio for the game.
- Programming the whole application.
- Preparing the documentation.

As can be seen in Figure 2.1, there is not a section for preparing the documentation because that is done simultaneously with other project tasks.

The project tasks have been distributed in the following way:

- Art: The first week is dedicated to make and gather the game arts. Regarding the menu, some icons and backgrounds are made with Gimp.

  With respect to the card art, the Spanish variant for the front cards is taken from the GNU operating system given that the arts have enough quality for the game and they can be freely used due to its GPL license. They have been slightly retouched for matching the game sizes via Gimp.

- Design: The applications were designed as two different ones in the beginning (client and server), but this is very unpractical for the users and it was changed for being only one.

  Also, the design of the applications should be almost identical for the user, and it will take two weeks for all the design to be finished.

- Audio: This is planned to do in one whole week, make a pair of simple seamless pieces can be easily done in this period of time.

- Programming: This is the most time and resources consuming for this project taking 3 months to complete it. It alone takes double time than the rest of the parts. So it is divided in more task and sub-task than the others.

  Programming an online multiplayer game is divided in the server application and the client application. In this project there is no online server, the server is one of the applications connected. Because of this, the server and the client must be done in parallel, this should take two and a half months for this.

  In the programming part, there will be three days to make the software design, and two weeks for the game base code, this will be made for the server application and for the client at a time.

  One month and one week for the game rules with the player turns, deck and game zones. This will be done only with the server part, the client will obey this rules because the authorization movement of the cards.

  Ten days for creating the network and to implement the communication protocol among applications. It will also be needed two more weeks to make the application handle the network events that allow the applications to synchronize.

  Finally, the remaining two weeks will be dedicated to create all the animations and effects needed for the game.

These tasks have been accordingly described in a Gantt chart (made via GanttProject) that shapes the schedule (see Figure 2.2). In it, it can be seen four big sections: design, art, programming and audio. The programming part is done simultaneously on both parts, the server and the client. There is no sense to make only one part and expect it will work with the other tasks pending to be done.

Type	Task	Sub-task	Sub-sub-task	Hours	
Documents	Technical Proposal			5	55
	Analysis and Design			7	
	Memorandum			35	
	Presentation			8	
Meetings				20	20
Art	Menu			6	15
	In Game cards			9	
Design	Server app			4	8
	Client app			4	
Programming	Server app	Software design		4	187
		Base game code		20	
		Game rules	Player turns	36	
			Deck	7	
			Game zones	12	
		Create network		15	
		Handle networking events		20	
	Client app	Software design		3	
		Base game code		20	
		Join network		10	
		Handle networking events		20	
	Animations & effects			20	
Audio	Music			10	15
	Fx			5	

Figure 2.1: Planning table

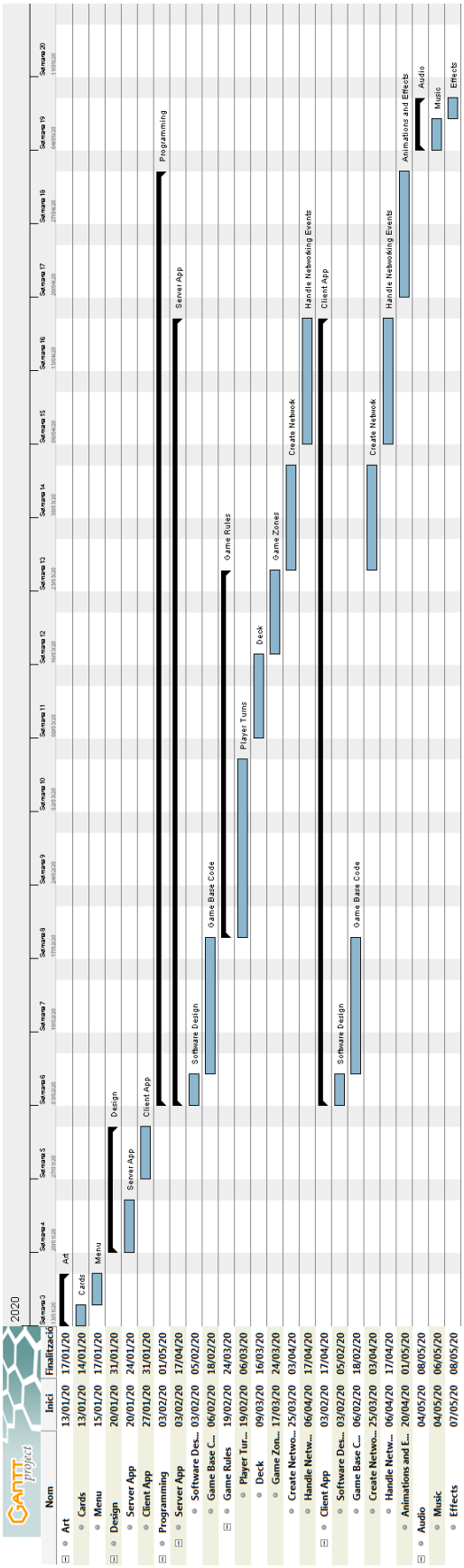

Figure 2.2: Gantt chart.

2.2 Resource Evaluation

The costs of the work that is going to be undertaken must be estimated in advance. The human and equipment costs must be quantified so that the work can be assessed and so that, in a real case, the economic viability of the work could be evaluated.

The average salary of a junior programmer in different companies in the industry is 23.480 €/year [7], so this project made in 300 hours should cost 3.355€ plus taxes in salaries.

Also there are some hardware and software to consider:

- Hardware:

  One computer, it does not need to be especially powerful, the app's graphical part is rather simple. The computer can be used for other projects too, but it is needed. Despite this, a computer is needed and it would cost 600 €.

- Software:

  Unity is free if the company's revenue in the last 12 months are $100k or less, if the revenue is between $100k and $200k it costs $40 per month (+177€ for this project), and if the revenue is greater than $200k the price is $150 per month (+664€ for this project).

  The IDE used is Rider, it costs $349 the first year (310 € for this project), but Microsoft's Visual Studio can be used too and it is free.

The total amount for this project will be 4.442€ including all the software, hardware and salary.

CHAPTER 3

System Analysis and Design

Contents

In this chapter we provide the requirements analysis, the system design and architecture of the proposed game, as well as, its interface design.

3.1 Requirement Analysis

3.1.1 Functional Requirements

A functional requirement defines a function of the system that is going to be developed. This function is described as a set of inputs, its behavior, and its outputs. The functional requirements can be: calculations, technical details, data manipulations and processes, and any other specific functionality that defines what a system is supposed to achieve.

The functional requirements are:

- When the user requests information about the assets available, all the assets must be returned divided into four categories: States, Objects, Commands and Events (see Table 3.1).
- When the user right-clicks on an empty zone in the bolt assets window and selects new asset's type, a new one of that category with the default name will be created (see Table 3.2).

- When the user holds down the Ctrl+button, a little red “x” will pop up next to all assets and properties allowing to delete them (see Table 3.3).
- The system will compile the assets when the user clicks the little green arrow icon in the Bolt Assets window or through the Bolt/Compile Assembly menu option (see Table 3.4).
- When the user clicks the “Server” button and starts a new game being the server, a new game room will be created and the player joins in automatically. The scene changes to the defined game (see Table 3.5).
- When the user request information about the game rooms available, they will be returned (see Table 3.6).
- When the user clicks on the button of the game room ID, the player will join that game room and the scene changes automatically where the server is (see Table 3.7).
- When the player drags a card across the board, the system will make the card move equally in all devices (see Table 3.8).
- When the player double taps where a card is, the card will change from face down to face up or vice versa (see Table 3.9).
- When the user requests the options available, they will be presented to them (see Table 3.10).
- The user can select the style of the buttons used in the menu (see Table 3.11).
- The user can select the back face of the cards used in the game (see Table 3.12).
- The user can change the volume of the music used in the game (see Table 3.13).
- The system will save automatically the options selected by the user (see Table 3.14).
- The system will load automatically the options saved, if they do not exist, a random ones will be used (see Table 3.15).

Input:	Available assets query
Output:	Available assets
The user requests the bolt assets available. When the user requests information about the assets available, all the assets must be returned divided in into four categories: States, Objects, Commands and Events.	

Table 3.1: Functional requirement «Bolt assets query»

Input:	The user add a new bolt asset.
Output:	New asset created.
The user right click on an empty zone in the bolt assets window and select new asset's type and a new one of that category with the default name will be created.	

Table 3.2: Functional requirement «New bolt asset»

Input:	Remove bolt asset.
Output:	Selected asset deleted.
If the user holds down the Ctrl button, a little red x will pop up next to all assets and properties allowing to delete them.	

Table 3.3: Functional requirement «The user deletes a bolt asset»

Input:	The user compiles all the assets.
Output:	A bolt.dll file will be created with all the assets compiled.
To make Bolt aware of the assets it is needed to compile them, this can be done either through the Bolt/Compile Assembly menu option or with the little green arrow icon in the Bolt Assets window. They both do the exact same thing.	

Table 3.4: Functional requirement «Compile assets»

Input:	In the game, the user creates a game server.
Output:	A game room is created and the user joins in.
In the game, the player clicks the "Server" button and starts a new game being the server, a new game room will be created and the player joins in. The scene changes to the defined game. Then they wait for start the game while other players can join in.	

Table 3.5: Functional requirement «Create game server»

Input:	Available game servers.
Output:	Game rooms for this game.
Each game has its own game rooms, when the player requests information about the game rooms available, this will be returned.	

Table 3.6: Functional requirement «Game server query»

Input:	In the game, the user joins a created game.
Output:	The user joins a game room.
In the game, the player clicks the button with the room id and joins in, the scene changes to the same where the server is.	

Table 3.7: Functional requirement «Join game»

Input:	In the game, the player drags a card.
Output:	The selected card moves in all devices.
In the game, the player can drag a card across the board and the card moves equally in all devices.	

Table 3.8: Functional requirement «Card movement»

Input:	In the game, the player double taps a card.
Output:	The selected card rotates face down or face up.
In the game, the player can turn down or up the cards by double tapping them. If it is face down it will turn face up, and if it is face up it will turn face down	

Table 3.9: Functional requirement «Card rotation»

Input:	In the game, available options.
Output:	The options available.
In the game, the player can request the options available and they will be presented to them	

Table 3.10: Functional requirement «Game options»

Input:	In the game, button background selection.
Output:	Button background selected.
In the game, the player can change the buttons style used	

Table 3.11: Functional requirement «Game buttons»

Input:	In the game, card game back.
Output:	Back card selected.
In the game, the player can replace the design on the reverse of the cards with alternate designs	

Table 3.12: Functional requirement «Game cards back»

Input:	In the game, volume.
Output:	Music volume changed.
In the game, the player has a slider for change the music volume or deactivate it completely. The volume changes in real time	

Table 3.13: Functional requirement «Game music volume»

Input:	None.
Output:	Saved options.
In the game, the system will save the options selected by the user automatically whenever they are set.	

Table 3.14: Functional requirement «Game save user options»

3.1.2 Non-functional Requirements

Non-functional requirements express how the system should be in overall, not what will be capable of. Some of them are:

- The system needs to be able to connect to the matchmaking server and the other devices.
- The network entities will be controlled only by the user who has authority over them.
- The network entities need to be updated in real time.

Input:	In the game, volume.
Output:	Music volume changed.
In the game, the options saved will be loaded automatically every time the application starts without the interaction of the user.	

Table 3.15: Functional requirement «Game load user options»

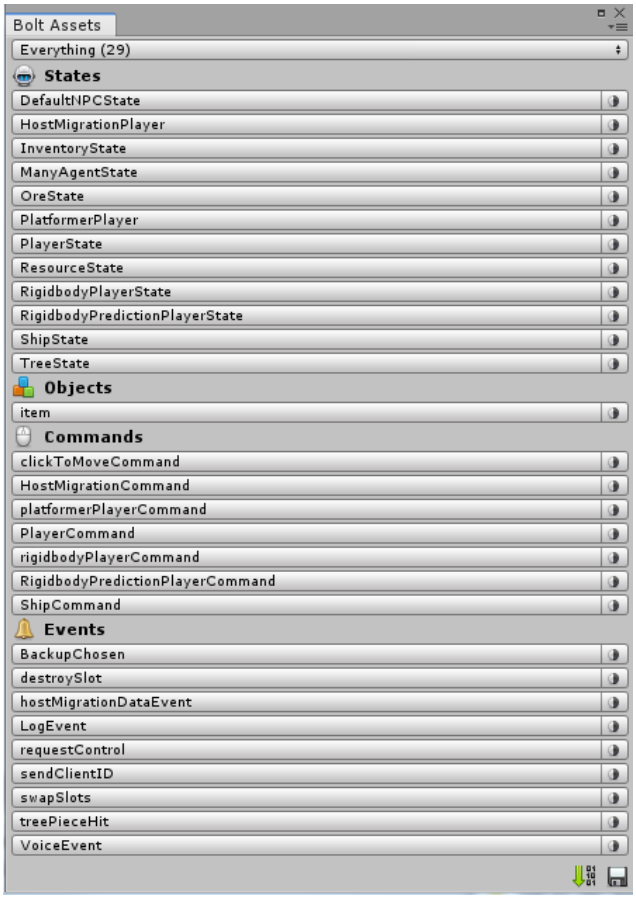

Figure 3.1: Bolt assets window

- If the network is unreliable, the system has to predict the entities movement smoothly.
- The application has to run in multiple operating systems: Linux, Windows, Mac OS X and Android.

3.2 System Design

The interaction with the system, activity diagram and class structure are represented with UML diagrams. These diagrams are made using the Dia application [2]. The system can be represented in different ways depending its behaviour, because of that there are different diagrams for each kind of representation:

- The application flowchart represents how the game is conducted (see Figure 3.2 and Figure 3.3). Two different flowcharts are made because there are two sides, the server and the client. They differ in the create the game or join a game part.

When the application starts, it shows the main menu. From the menu, the user can choose to go to the options or start a game (for the server) or join a game (for the client). In the options, it is possible to choose the sound volume, card back faces and the button style in the menu. When starting a game as a server, it is needed to select the game you want to play, and then, the client can join the game created. After that, the game can start. The client can not select the game to play, when joining a game it is selected the one the server player selected.

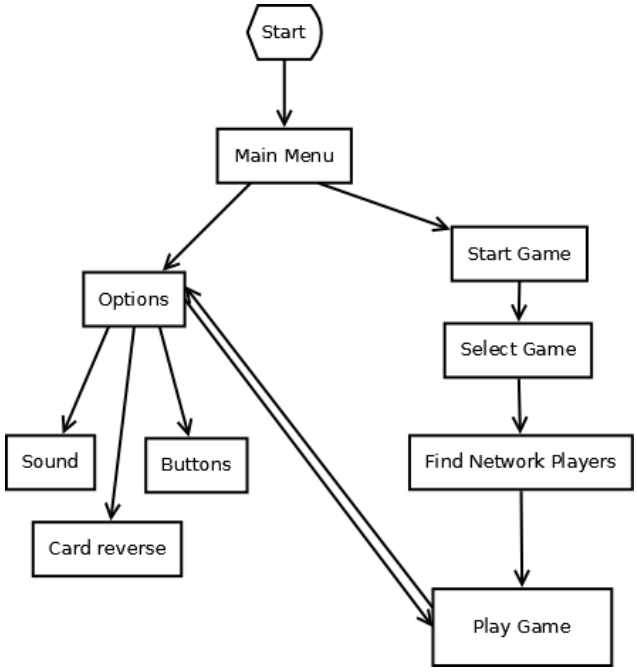

Figure 3.2: Server flowchart.

- The use case diagram (see Figure 3.4) represents in a UML diagram how the user interacts with the game.

  The user can select to create a game room for playing, this implies other users can join your game. Creating a game room also implies a game start. Starting a game creates the cards and moves them to form a pile of cards. The player can also move any card on their own. It is possible for the player to rotate the cards.

  The user can join a game room instead of creating one, this will make the user wait for the server player to start the game.

- The class structure diagram (see Figure 3.5) represents in a UML diagram how the internal structure is made and its inner relationships.

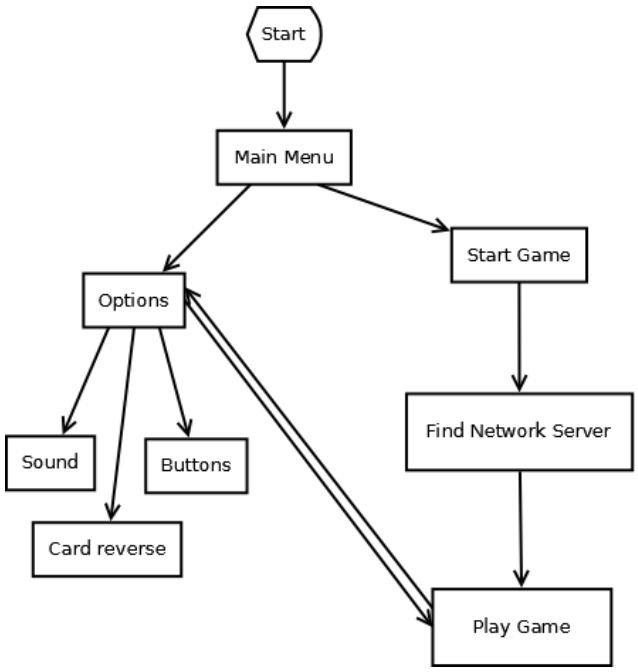

Figure 3.3: Client flowchart.

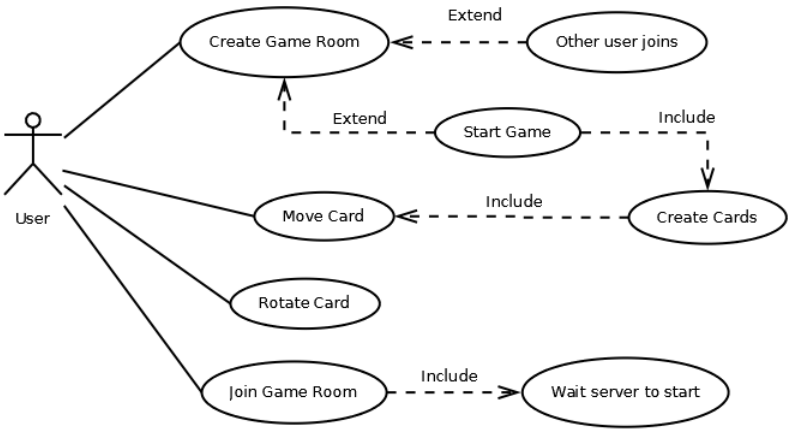

Figure 3.4: UML case diagram.

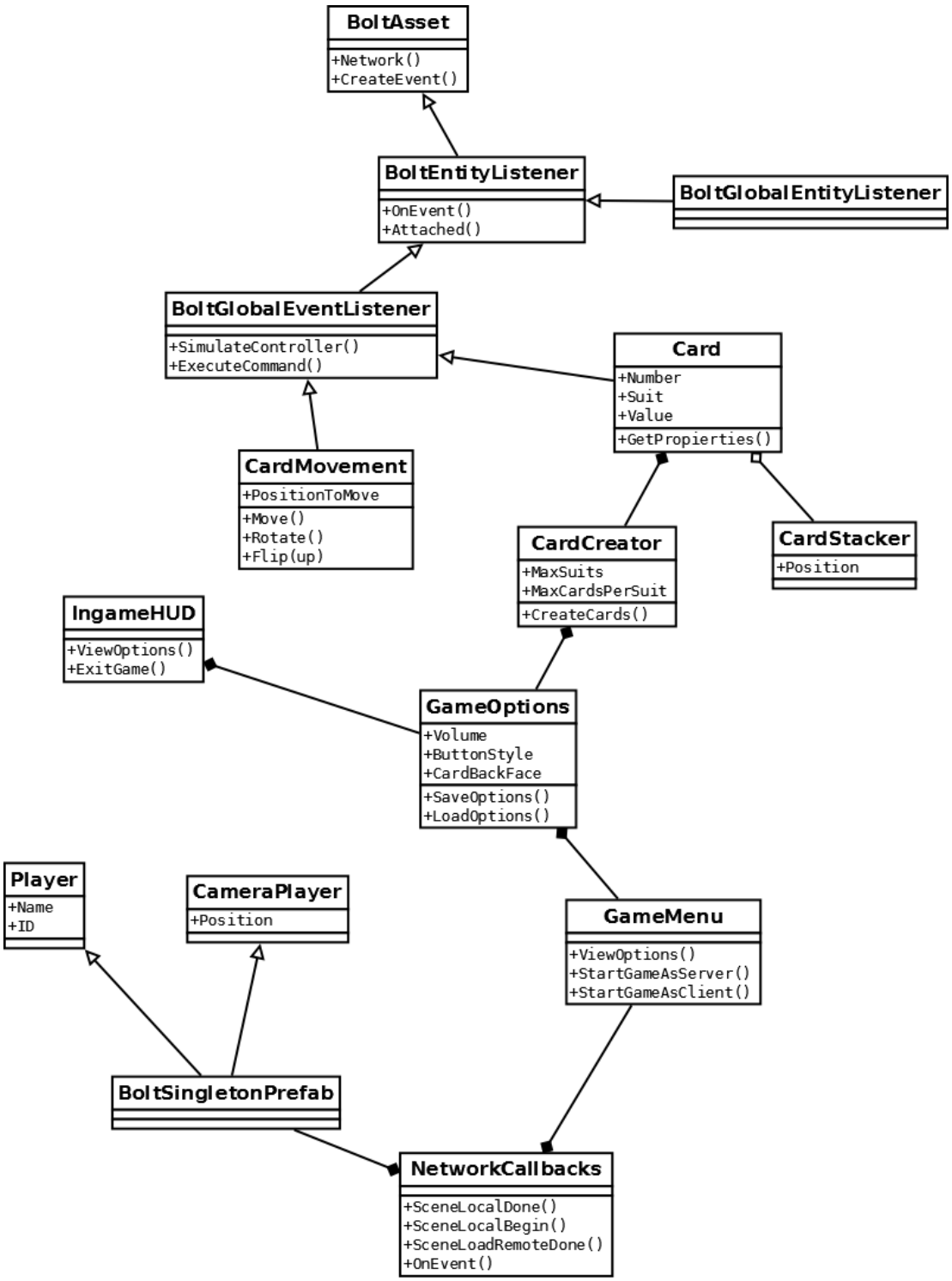

Figure 3.5: Class diagram.

- The activity UML diagram describes how the objects are used and the relationship between the different activities. In the diagram (see Figure 3.6), we can see the flow in the application since it starts. In the beginning, there is the main menu where the user can select to be a server for starting a new game, being a client to join a created game or to go to the options page.

  In the options page, the user can select the menu buttons style, the music volume and the card back faces in game. From the menu, it is possible to create a game, after this a new game room is created and joined. Now is possible to start a game. While joining a game as a client, a network search is performed to find the game rooms available. The client user joins a game room they found and the game starts.

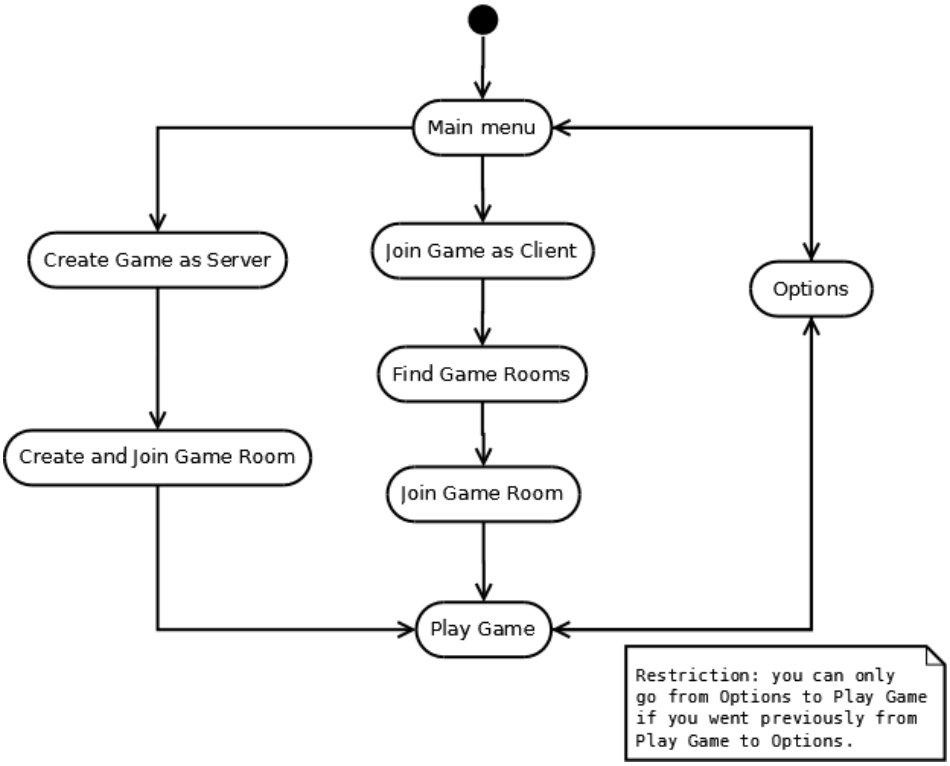

Figure 3.6: Activity Diagram.

3.3 System Architecture

The aspects of the system needed to run this game are:

- A system with a pointing device like a mouse or a touch screen. This is needed to interact with the menu buttons and with the cards for moving them.

- Internet connection. Be able to connect to the server and preferably be able to do p2p connections. The connection between the devices is done through internet via a server, and then via p2p or with the server if it is not possible to be established via p2p.
- The minimum OS requirements for playing any Unity application are Android 4.3 or up, Windows 7 SP1 or up, macOS Sierra 10.12.6+, almost all Linux distributions with kernel 4.4+ (like Ubuntu 16.04) [14].

3.4 Interface Design

The game has a rather simple interface, the graphical elements do not have to interfere with the gameplay. There are tree main interface designs:

The main menu has the game logo on top and three buttons for accessing the different sections (see Figure 3.7).

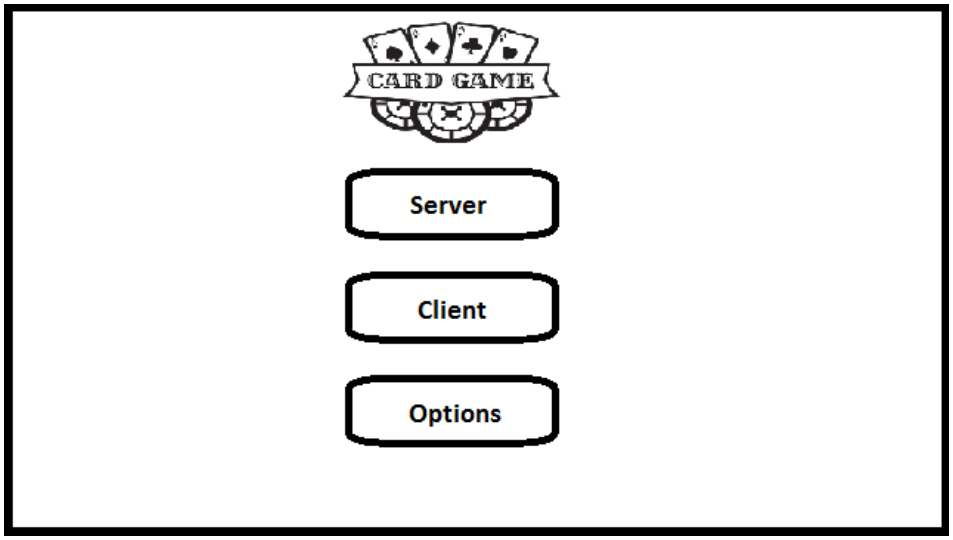

Figure 3.7: Main Menu Design

In the *options* section (see Figure 3.8), on the top we can see the different button styles to select the one that has to be used, a preview of each one available are displayed there.

In the middle, there is a slider controlling the volume of the music. On the left it will be muted and on the right it will be loud. The regulation is almost continuous.

The different card back faces are on the bottom, there is possible to select the art that will be used in the game.

The in-game interface (see Figure 3.9) has to be the most simple and bare possible, but there are some elements that need to be there, like a gear on the top right corner

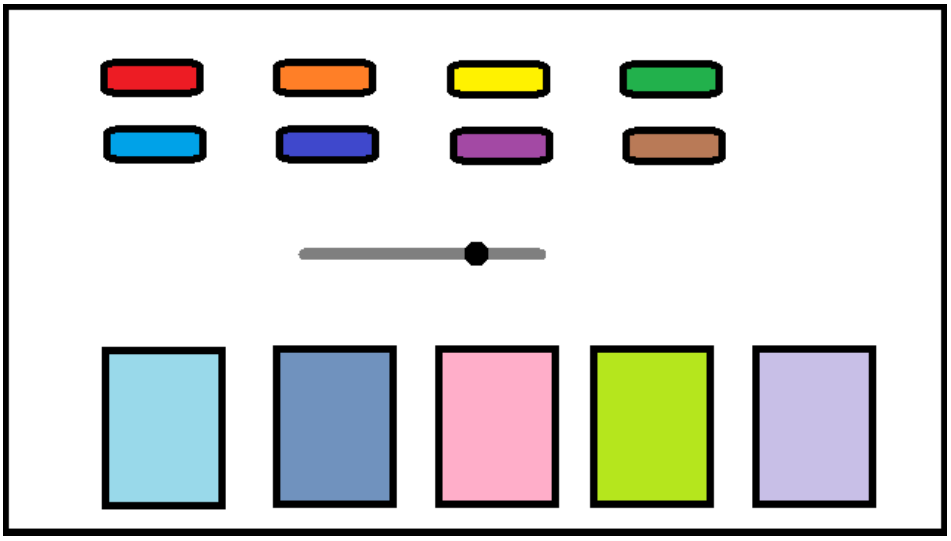

Figure 3.8: Options Menu Design

to accessing the options in the game. Also, there is a button to move the camera and change the view from the player's cards to the cards in common or in the table, this button will be on the top left corner. On the top mid will be the score if the game has one.

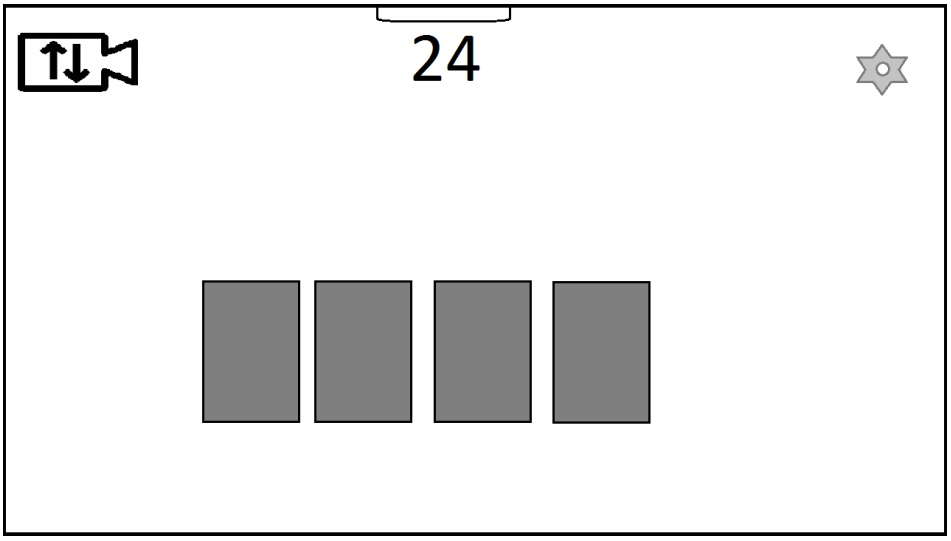

Figure 3.9: Game Interface Design

Chapter 4

Work Development and Results

Contents

In this chapter is where all the developed work is explained. The work in a video game has different aspects to cover, these aspects are described below. Also, there is an explanation of the result achieved in this project.

4.1 Work Development

4.1.1 Art

Here we can distinguish two parts, the menu and the Heads Up Display (HUD [15]) icons, and the card frames:

- **Icons**: The icons in the menu must be simple and easily understandable by anyone just by looking at them. A simple cross says the same than "click here for close this" and does not overload the screen (see Figure 4.1). Familiar icons are used to accomplish common tasks.

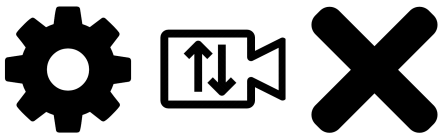

Figure 4.1: Icons.

The buttons have a selectable style, all of them not too dark so the black characters on the buttons are easily read.

- **Card faces**: The cards have two sides; on the front we have the art, and it can be the spanish style with 4 suits and 10-12 cards per suit, or the french style with 4 suits and 10-13 cards per suit (see Figure 4.2).

Figure 4.2: Different front faces.

On the back face there are some different card frames (see Figure 4.3), the one desired can be selected in the *options* menu.

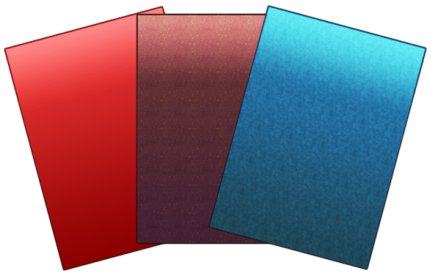

Figure 4.3: Different back faces.

4.1.2 Design

The application needs a design, it was made following the flowcharts shown in the last chapter (see Figure 3.2, figure 3.3 and figure 3.9).

The menu has the aspect planned, the logo on top and three buttons for each one of the operations (see Figure 4.4).

Figure 4.4: Final version of the main menu.

The options menu has three different zones: on the top is where the button style is selected, in the mid there is a slider for control the music volume, and on the bottom is the back card image selection (see Figure 4.5).

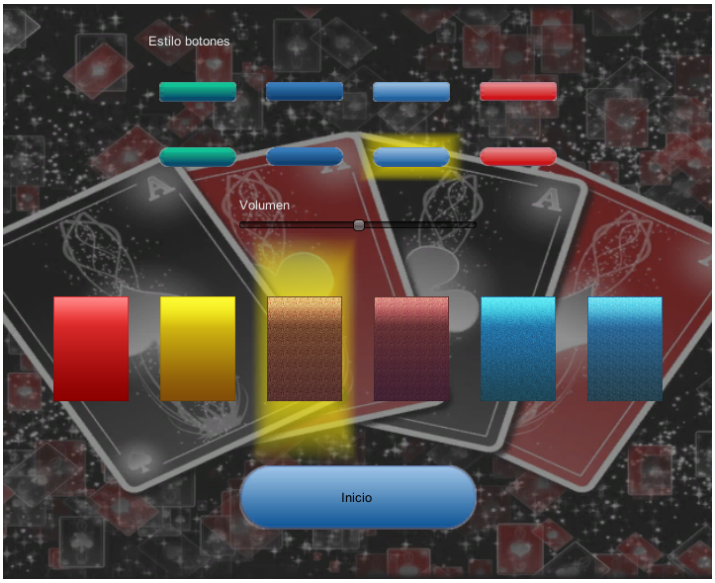

Figure 4.5: Final version of the options menu.

4.1.3 User Experience

As Celia Hodent says in her study [9], the user experience (UX) entails a person's perceptions and interactions with a product or software (such as a video game) and the satisfaction and emotions elicited via this interaction. UX overall refers to an overarching discipline focused on the evaluation and improvement of users' experience of a given product or software in development.

The user interface and the user experience are closely linked. This game is a puzzle-like kind of game, the user interface needs to be simple for the user being focused in the game.

This game tries to emulate a real-life game. The most important aspect of the emotional game design is the "game feel", the tactile sensation of interacting with the virtual cards. The feeling of moving the cards is smooth and natural while dragging them (see Figure 4.6).

The heuristics developed in this game take into account the limitations of the human brain in perception, attention, and memory, being the interface and the controls the most simple possible, or if it has some doubt on how to do anything, the first guess from the user is the right one.

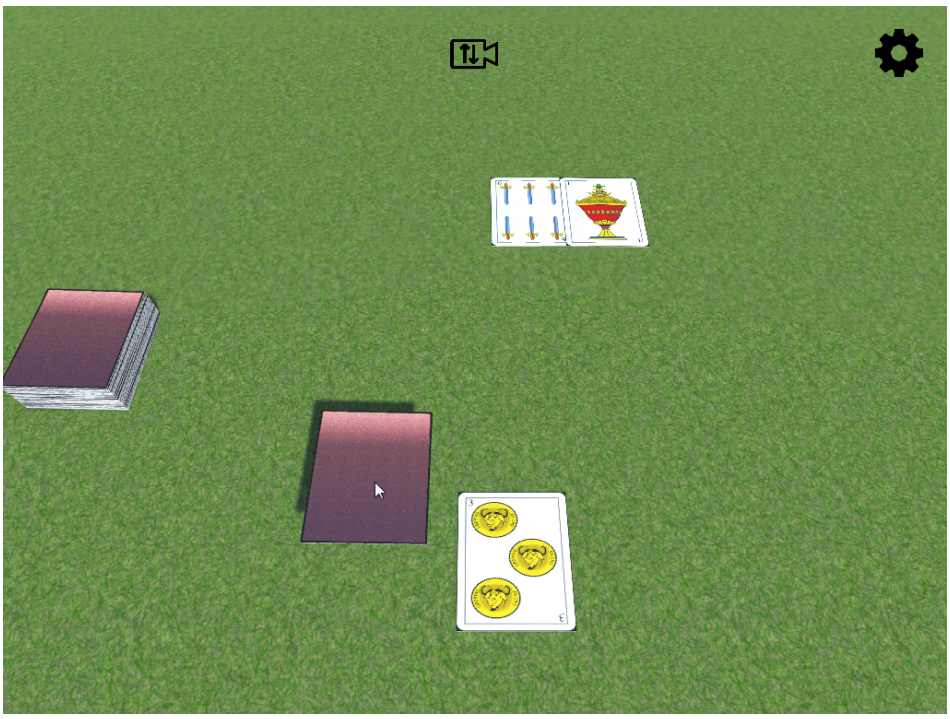

Figure 4.6: Game feel. Moving a card.

4.1.4 Programming

This is the part what took the most of the work, and the most extensive one. For convenience, it is going to be split into parts below.

Menu

The menu part is done with the GUILayout class in Unity as it is a very efficient way to implement this feature [12].

Indeed, this feature permits us to use the application as a server or as a client by doing the right button labelling (see Figure 4.7). In the server part, there is a selection of games available, and after choosing one, a game room is created and the game waits for the clients to connect (see Figure 4.8). In the client part, there is a list of games created where the player can join the room to play in it (see Figure 4.9).

Figure 4.7: Main menu.

In both parts, server and client, there will be a temporal information box that tells how many people is connected, including some information about the last connected or disconnected client (see Figure 4.8).

It is also implemented a section for the options where the user can set the different customizations, like the button style, card backfaces or the music volume (see

Figure 4.8: Server connecting.

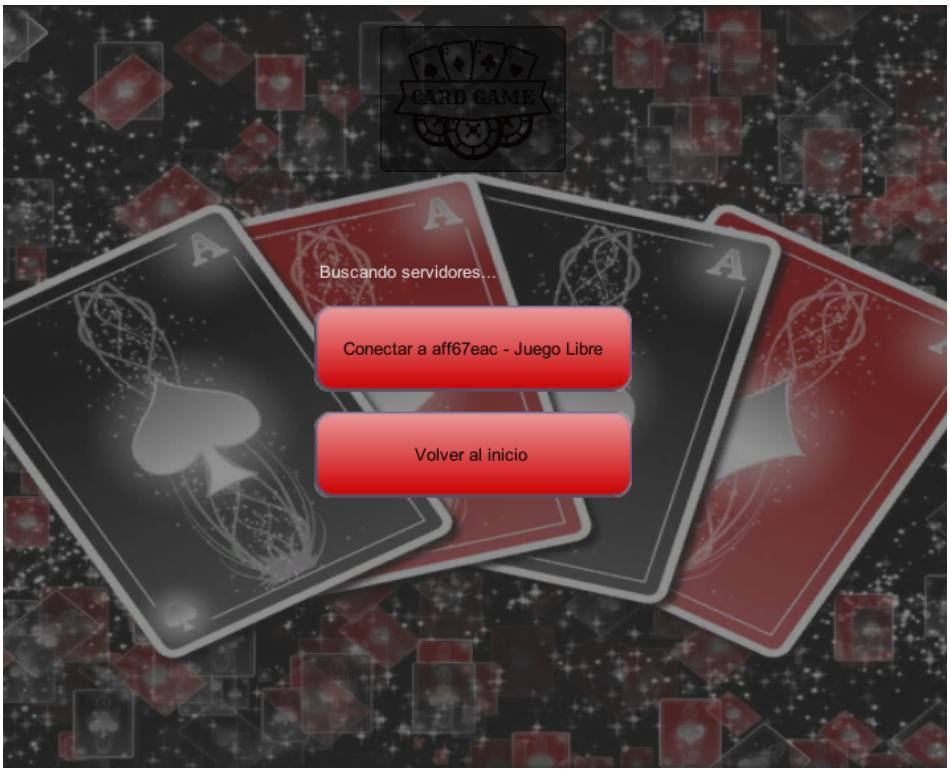

Figure 4.9: Client connecting.

Figure 4.10). These options are locally saved when the user goes back to the main menu and loaded when the application starts (if they exist). This is done using the Unity PlayerPrefs [13] because it has support for multiple platforms in an easy way to add the saved options for all the players in the games.

Figure 4.10: Options.

Network

When an object is needed to be replicated over the network, we need something called *assets*. The *assets* are pre-compiled in the editor and, after doing it, that network entity can be added to the game objects.

When compiling, Bolt will go through all of the prefabs and other Bolt-related assets and compile a very efficient network protocol for them, which is then stored inside of the *bolt.user.dll* assembly which you can find in *Assets/Photon/PhotonBolt/assemblies* folder.

Assets are broken down into four categories:

- **States**: these are the meat of most games built, they allow the programmer to define features like the player name, health, transform, animation parameters, etc. (see Figure 4.11).

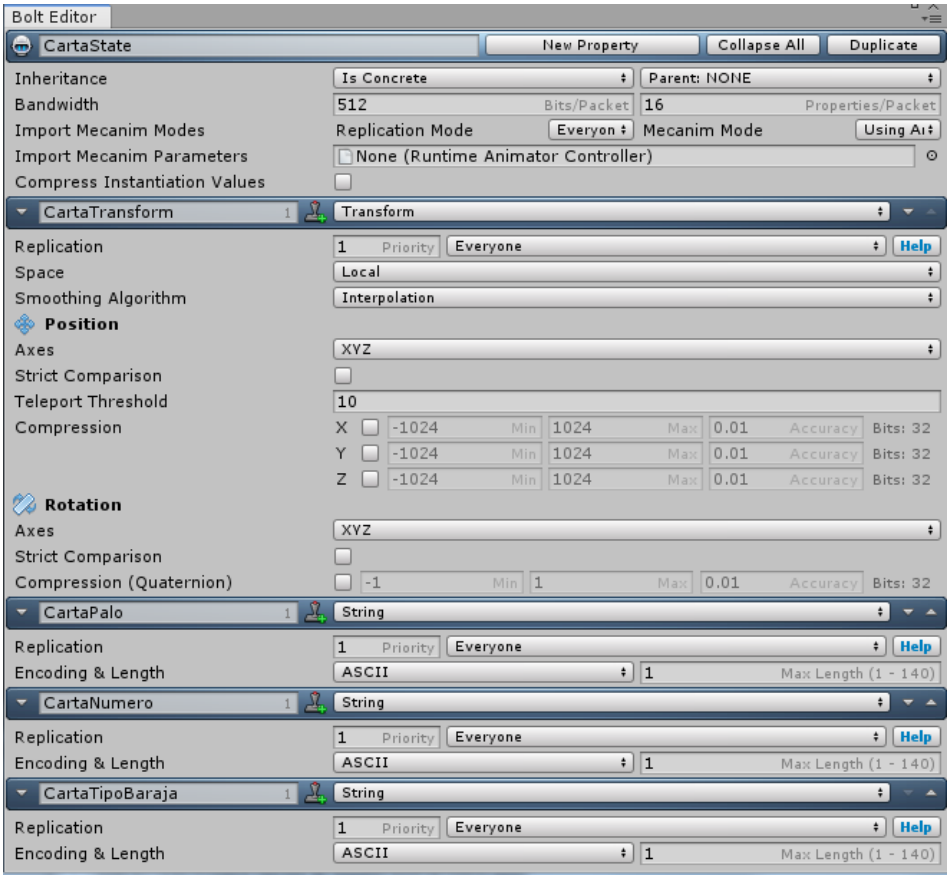

Figure 4.11: State window.

- **Objects**: objects are a way of logically grouping several properties in the same way a programmer does with a C# class. As an example, if an inventory has to be implemented for a player in an RPG game, an Object called *Item* can be created and assign it properties like *ItemId*, *Durability*, etc. Then, the *State* creates an array of *Item* objects (see Figure 4.12).

- **Commands**: these are used exclusively for the authoritative features and are used to encapsulate user input and the result of applying the input to a character in the game (see Figure 4.13).

- **Events**: these are simply messages. Events allow the programmer to much easier decouple the different parts of the game from each other, as it can have several listeners to a single event active at the same time (see Figure 4.14).

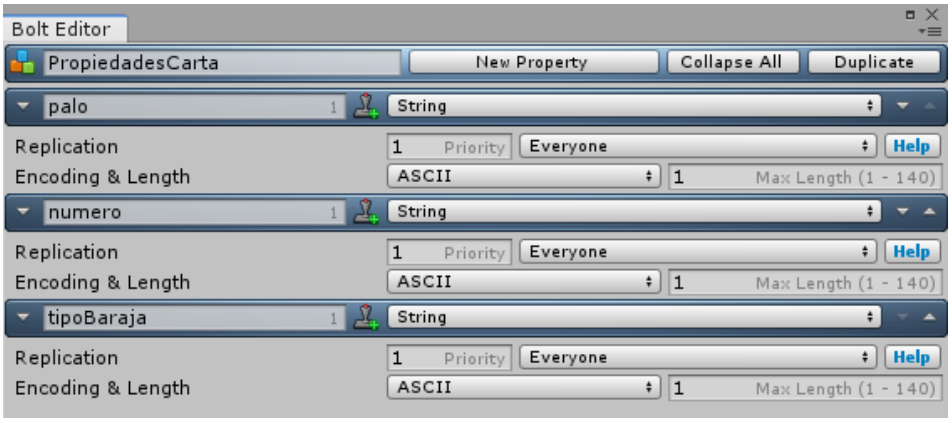

Figure 4.12: Object window.

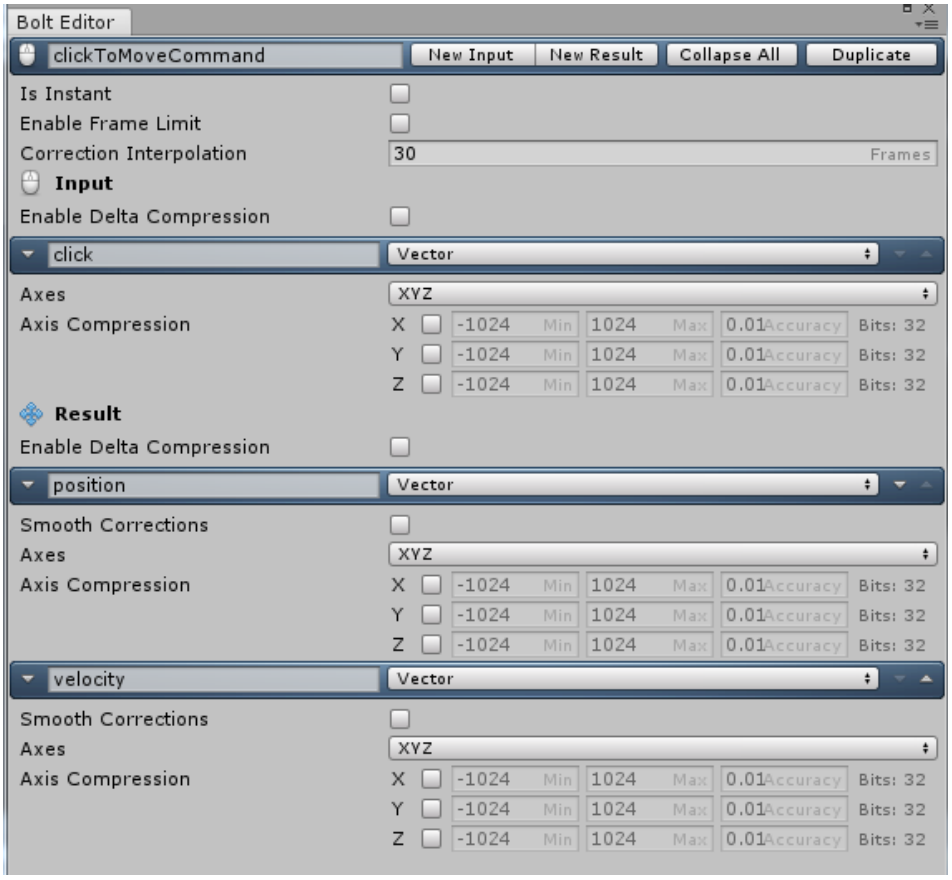

Figure 4.13: Commands window.

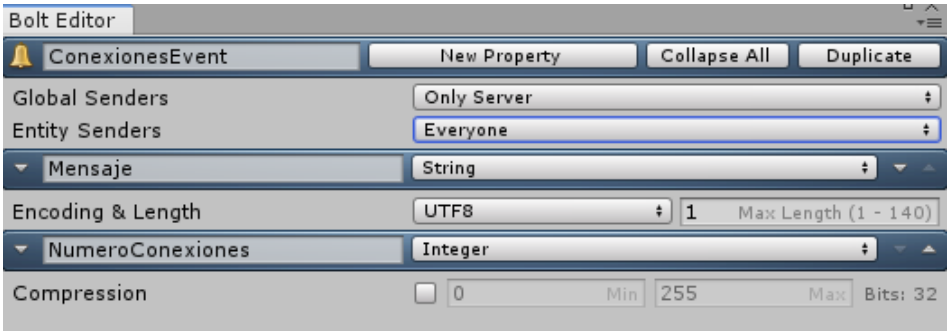

Figure 4.14: Events window.

There is another particular kind of script, the *NetworkCallbacks*, it inherits from *GlobalEventListener*, and it is not attached to any game object. It is used to handle events over the network like when someone is connected or when the scene is fully loaded in all the clients. It can be accessed only from the server or from a client, this is the only script that specifies the behaviour from the server or the client differently.

Server callbacks in C#

```
1  using UnityEngine;
2  using System.Collections;
3
4  [BoltGlobalBehaviour(BoltNetworkModes.Server)]
5  public class NetworkCallbacksServer : Bolt.GlobalEventListener
6  {
7      private int numeroConexiones = 0;
8
9      public override void SceneLoadLocalDone(string scene)
10     {
11         if (scene == "Juego_Libre")
12             GameObject.FindGameObjectWithTag("CreaCartas").GetComponent<CreaCartas>().CrearCartas();
13
14         if (scene == "Menu")
15             BoltNetwork.Shutdown();
16     }
17
18     public override void Connected(BoltConnection connection)
19     {
20         var log = ConexionesEvent.Create();
21         log.Mensaje = string.Format("Conectado_{0}", connection.RemoteEndPoint);
22         log.NumeroConexiones = ++numeroConexiones;
23         Debug.LogFormat("Nueva_conexión_desde_{0},_ahora_hay_{1}_conexiones", connection.RemoteEndPoint,
24             numeroConexiones);
25         log.Send();
26
27         if (FindObjectOfType<GuiIngame>().estado != GuiIngame.Estado.ServidorEsperandoEmpezar)
28             connection.Disconnect();
29     }
30 }
```

The cards have almost all kind of assets included:

- They have States. A state is needed to keep the position and other parameters like the suit, value... synced.
- They have Events. We need an event to send the card to another place, or to flip it. A simple message to all the other clients will do this, we do not need to waste network bandwidth for this.
- In the events, we can choose two kinds of senders, global sender or entity sender. The global senders are received by all the listeners, and the entity sender is received only by the synced entity on the network. Also, in this kind of senders, it is possible to choose who can do the event: the controller, the owner, nobody or everyone.
- The events in the cards trigger a coroutine for doing the animation. This is done like that so it does not interfere with other actions done by the local player.

Coroutine for moving cards in C#

```
/*      Coroutine in C# */

IEnumerator Moviendome() {
        trasladando = true;
        Vector3 v3inicio = transform.position;
        float TiempoInicio = Time.time;
        float tiempoRelativo = Time.time - TiempoInicio;

        while (tiempoRelativo <= tiempoTranslacion) {
                tiempoRelativo = Time.time - TiempoInicio;
                transform.position =
                        Vector3.Lerp (v3inicio, posicionAMover, tiempoRelativo/tiempoTranslacion);
                yield return 1;
        }
        trasladando = false;
}
```

On the other hand, we also work with *commands*. The *commands* are meant to handle inputs from the user and the result of that input. However, this only works for the server or for the client that controls that entity and we want to make everyone be able to move all the cards, so we need to request the authority of the entity every time we are going to do a command and we do not have it [3]. This can be done with an entity event sender to the server. Having the authority, the server accepts the server simulation for this entity. See Figure 4.15 for a visual flow of what happens when someone tries to move a card.

While having control of the entity it is possible to drag the card across the board and replicate the movement in all the other devices like magic. With commands, we implement client-side prediction and correction of the wrong prediction.

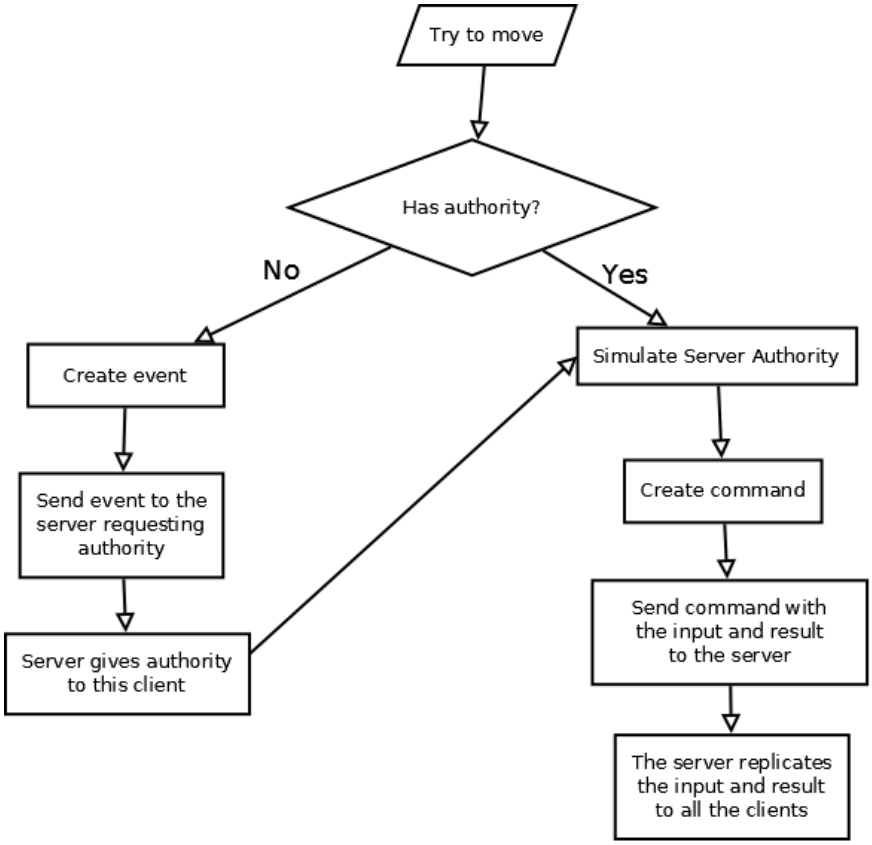

Figure 4.15: Card movement request flow.

Since the movement does not have to be abrupt and we are facing the asynchronous network, also there is a smoothing algorithm made by interpolating the initial position given and the final one.

The network callbacks are used in this game for keeping a record of connected devices. When someone connects or disconnects, it is sent an event to this class, it sends an event over the network informing the connection and shows a temporal message on the screen stating the new connection or disconnection [4]. This mechanism is also used for spawning local-only entities, like the camera or the player object.

The local camera and the player object are special game objects, there is only one at each client and it is not synced with the other clients. By having a different one for each game, we can not just throw the game object to the scene, it will be all the same in all the devices. The game object needs to have a script which inherits from *BoltSingletonPrefab*, so the network callback can know what game object to instantiate (remember it is not attached to any game object and it is not possible to a serialize field

or public variable here).

The options in-game are accessed via a button in the top right corner, it is shaped like a gear, a universally understandable icon for options. Here we have the same options that are shown in the main menu: change button style, card back faces, and the volume of the music.

There is only one difference here, the button for going back to the main menu is replaced by one to exit the current game (see Figure 4.16).

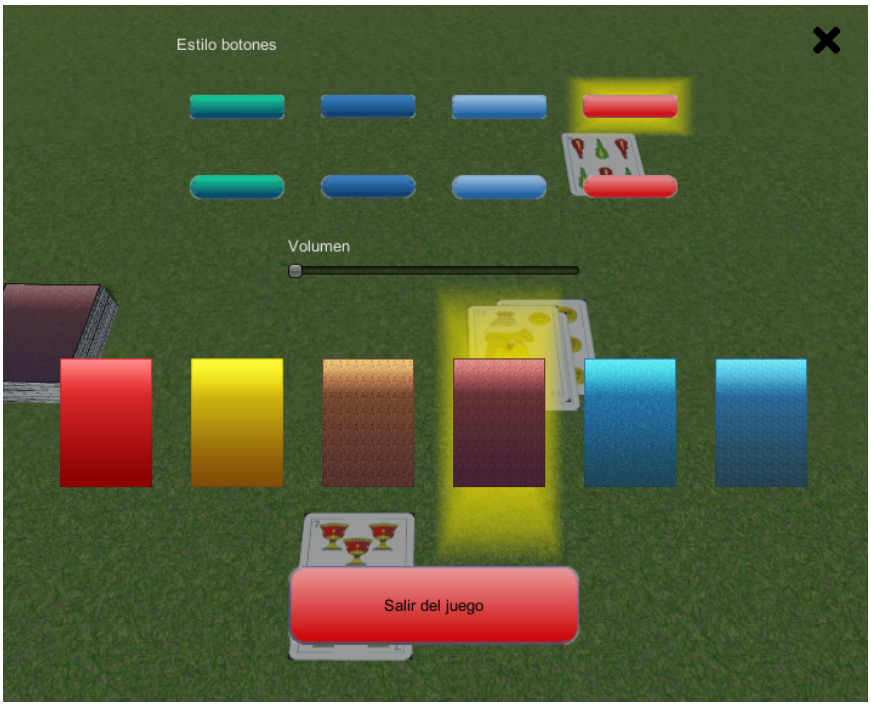

Figure 4.16: Ingame game options.

4.1.5 Audio

The music in video games does not have the same structure as the ones in the movies or the usual songs we listen to. Usually, in the music, there is a beginning or introduction, something in the middle like the theme and maybe repeated once with a different tone, and the end or coda where there is a conclusion.

In video games the beginning is the same, it is started playing when the game starts or when the user does something, but there is not an end, the game can take longer or shorter depending the actions taken by the user so it is needed to loop the middle part indefinitely till the game finishes. The middle part needs to fit seamlessly from the end to the beginning so there is not any gap or cut while playing (see Figure 4.17).

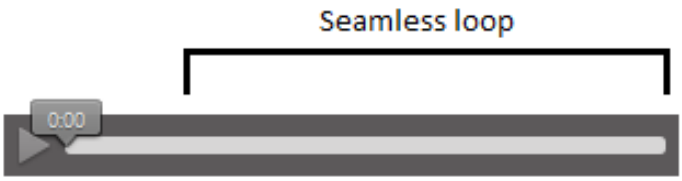

Figure 4.17: Music basic structure.

If there is a need for changing the music for something more exciting or relaxed, it is needed to do the music in layers and only activate the needed ones.

4.2 Results

The initial schedule was quite inaccurate, as the implementation of the network protocol and the objects' synchronization were very time-consuming tasks, which mostly took the same time as the implementation of all the remaining features of the game.

The menu (the first module it was made) is a great work of customization with the different styles of the buttons, card faces, auto-save and load feature for the user options, etc. However, in the end, it was possible to finish a playable multiplayer game, not just a proof of concept. There is some future work to extend this game with new features, such as a custom real game (see Figure 4.18).

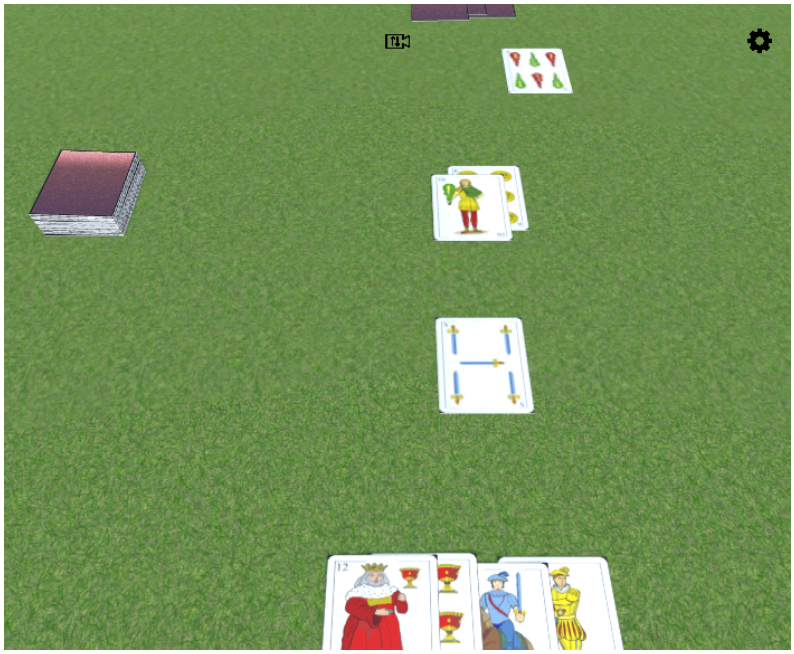

Figure 4.18: Game while playing.

Chapter 5

Conclusions and Future Work

Contents

5.1 Conclusions

We chose to create this video game as we wanted to learn how online multiplayer games work, not only by following a tutorial or by recycling code done by others, as all the multiplayer examples found have an avatar for the local player, that only the local player can control. Making this game means it is needed to understand how the inner network works and adapt it to the peculiarities of this card game.

At first the implementation of the game features were coded quite flawlessly, till the networking part appeared. A simple movement did not work on all devices and the documentation available on the internet was not sufficient to debug the game each time a problem appeared, also learning curve was very steep at the beginning. Indeed this task was very frustrating, and it took lots of days trying to implement a feature that at first seemed much simpler. Also when we did some progress, we were unable to show the advances to the tutor because they were closely related to the code, and did no show anything visible.

The most important thing we learned through this project is: online multiplayer is not easy to implement and it takes the same time than programming all the rest, so if it is needed to add online multiplayer to a game, it will take double time to develop.

The code is hosted on github [1] publicly available. Everybody can check it and see how it is made.

5.2 Future work

This project can be extended in the following ways, for example:

- New games can be added as modules, each one with its own rules.
- Voice chat, so the players can communicate just talking.
- Server migration. If the server disconnects, another client assumes the server role and the game does not need to finish.

Bibliography

[1] Jose Angel Buforn. Project repository. https://github.com/JoseAngelB/TFG2020. Accessed: 2020-02-28.

[2] Svitozar Cherepii. Dia programm. http://live.gnome.org/Dia. Accessed: 2020-02-28.

[3] Exit Games. Bolt entity ownership. https://doc.photonengine.com/en-us/bolt/current/in-depth/entity-ownership. Accessed: 2020-02-28.

[4] Exit Games. Bolt network callbacks. https://doc.photonengine.com/en-us/bolt/current/reference/state-callbacks. Accessed: 2020-02-28.

[5] Exit Games. Photon engine. https://www.photonengine.com/en-us/Photon. Accessed: 2020-02-28.

[6] Exit Games. Photon engine, creating a game. https://doc.photonengine.com/en-us/bolt/current/demos-and-tutorials/advanced-tutorial/chapter-1. Accessed: 2020-02-28.

[7] Inc. Glassdoor. Average junior programmer salaries. https://www.glassdoor.com/Salaries/junior-programmer-salary-SRCH_KO0,17.htm. Accessed: 2020-02-28.

[8] Don Glover. Unet deprecation faq. https://support.unity3d.com/hc/en-us/articles/360001252086-UNet-Deprecation-FAQ. Accessed: 2020-02-28.

[9] Celia Hodent. *Cognitive Psychology Applied to User Experience in Video Games*, pages 1–6. Springer International Publishing, Cham, 2016.

[10] Discord Inc. Bolt discord server. https://discord.gg/0ya6ZpOvnShSCtbb. Accessed: 2020-02-28.

[11] The GIMP Team. Gimp homepage. https://www.gimp.org/. Accessed: 2020-02-28.

[12] Unity Technologies. Guilayout reference. https://docs.unity3d.com/ScriptReference/GUILayout.html. Accessed: 2020-02-28.

[13] Unity Technologies. Playerprefs reference. https://docs.unity3d.com/ScriptReference/PlayerPrefs.html. Accessed: 2020-02-28.

[14] Unity Technologies. Unity requirements. https://docs.unity3d.com/Manual/system-requirements.html. Accessed: 2020-02-28.

[15] Wikipedia. Heads-up display (video games). https://en.wikipedia.org/wiki/HUD_(video_gaming). Accessed: 2020-02-28.