Once Upon A Card
Technical Design Document

Index

Overview

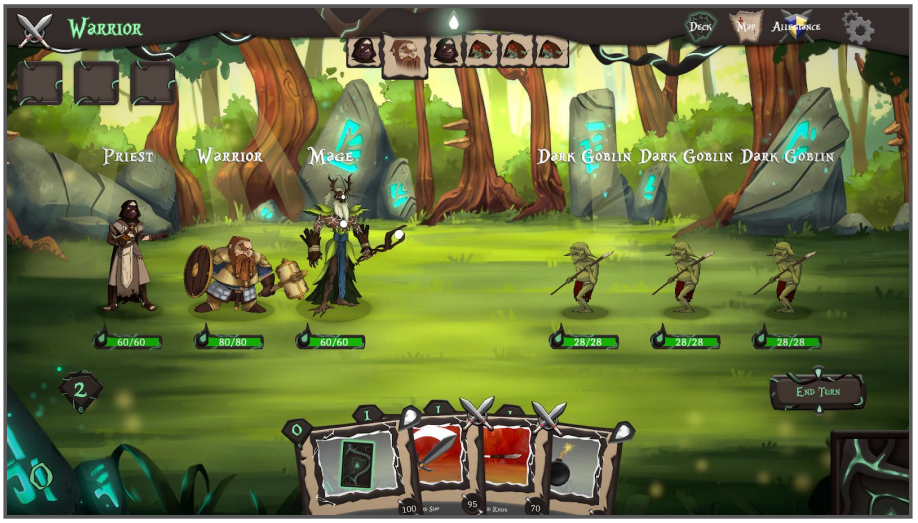
Game Summary

Once Upon A Card is a **2D online cooperative card game** up to **4 players**.
It has the particularity to possibly have one player being a traitor changing his objective to killing the 3 others.
This game contains **4 main phases** which are :

Path

Players have to vote for which path to choose

Combat



Inspired by *Slay The Spire*, fight NPCs or players by drag and dropping cards on your target.

Campfire

Inspired by *The Werewolves of Millers Hollow*, players can make anonymous actions on each others during night which will resolve simultaneously on the morning.

Textual Events

Textual events in which players will have to vote for the wanted resolution (ex : Heal your camp or fight).

System Requirements

Computer running Windows 7+
Mouse/Keyboard
Graphic Card

Technical Risks / Challenges

Being a **card game**, Once Upon A Card will need a **strong and clean template** to create **ability easily with a large possibility of effects**.
Moreover, **combat** being **centric** in the game, it should be able to handle **various amount of NPCs and Players**.

In the idea of **board game**, **UI** is a **major part of player interactions** and should be **intuitive**, with **enjoyable feedbacks** and **animations**, and an **appealing aspect** which can be enhanced with some shaders.

Finally, the **most risky part** will be the **networking**, all **game phases** should be **synchronized** and the **whole game code** will have to work whatever the players **latency**.

Tools

Engine Used

Unity 2018.3.2f1

3rd Party Tools / Assets

- Dotween
- Photon Bolt
- 2D IK

Versioning

Using Sourcetree and GitLab

Code Overview

General Architecture

The game is composed of **Players** all managed by a **Game Manager** which will handle all main game methods. This **Game Manager** will be created once the player create or join a **Room**.

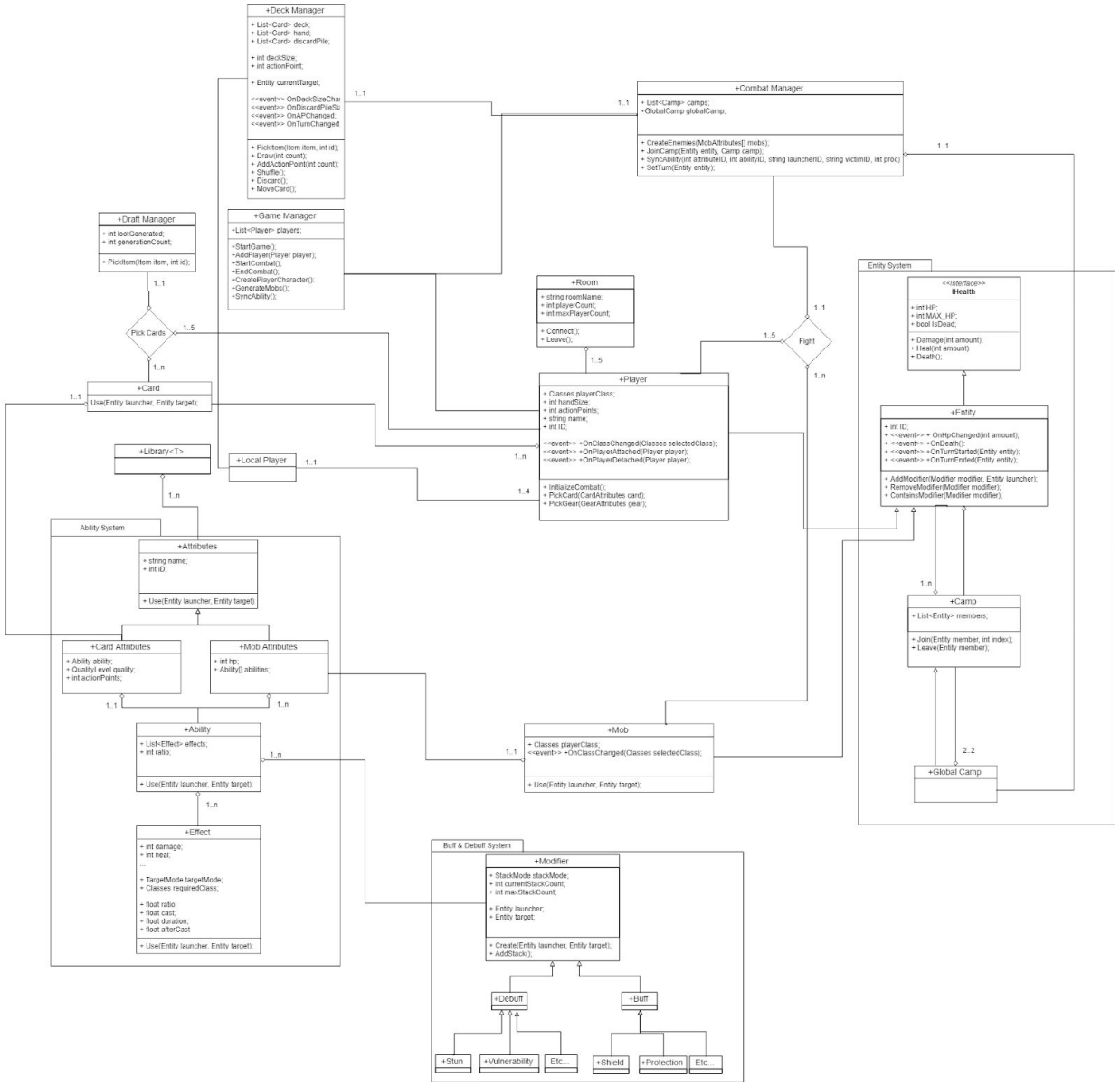
Card are composed of **Card Attributes**(scriptable object) which will store all data required and **Abilities** which are composed of all combat **Effects**.

Once in **Combat**, a **Combat Manager** will make players create their **Entities** and will create **Mobs** and will fill the two **Camps**. Every method related to card in combat (like draw, discard, etc…) are managed by a **Deck Manager**.

A **Travel Manager** will handle map generation and contains all method relative to the **Path**.
In order to ease the synchronization, important game elements are inheriting from **Attributes** which are stored in **Libraries** which is a generic class.

In terms of networking, the **Game Manager** will send all main **Bolt Events**, then the **Global Event Manager** will receive them and call methods depending on the **Bolt Event** received.
Network Callbacks is a network class here to know when the local player has loaded a scene and register **Bolt Protocol Tokens**, when **Server Callbacks** manage players connections.

Class Diagram



https://drive.google.com/file/d/1f9RGtOFiyf2NVjyFULx6IF90nN-YaPqc/view?usp=sharing

Entities Description

Player : contains name, ID, classes, stats and method to create **Entity**. It's a **Bolt Entity Event Listener** and the network representation of the player client as server.

Game Manager : Contains all main game methods and manage all the players. it's also a network class which will send all network events if required.

Combat Manager : Manage everything related to combat like **Camps**, **Entities** creation, **Abilities** synchronization, and turns.

Deck Manager : Manage cards in combat, contains method to draw, discard, move a card, shuffle and use a card based on **Player**'s action points.

Entity : Representation of living things, contains health, and methods to take damage, heal and **Modifiers** management.

Card : Read and display card using **Card Attributes.**

Card Attributes : Store card data and an **Ability**.

Mob : Read and display mob using **Mob Attributes.**

Mob Attributes : Store mob data and its **Abilities.**

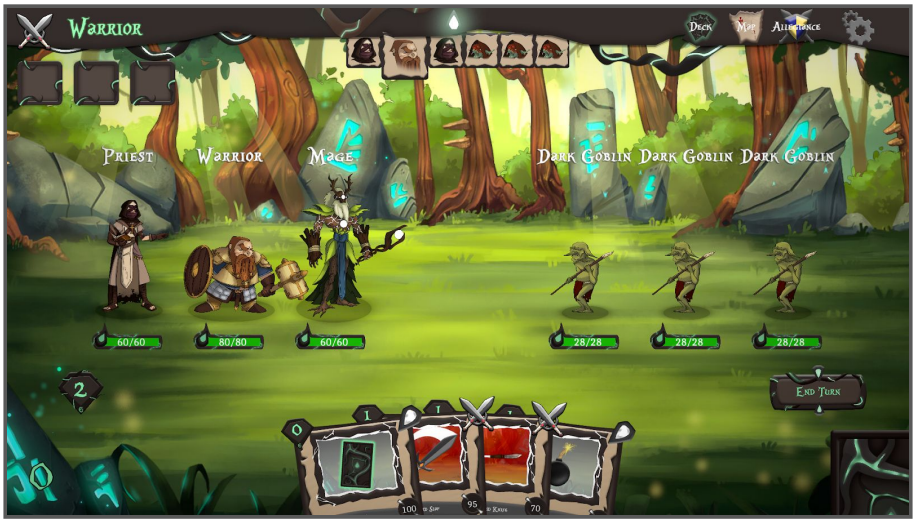
Ability : store a list of **Effects** and others data like hit ratio or animation to use.

Effects : store all data relative to combat like damage, target mode, **Modifiers** application etc and **Game Events.**

Library : Generic class storing **Attributes**

Technical Features

Combat



Being the main phase of the game, combat oppose a **Camp** to another one.
Camps are entities containing others entities called **members**, all camps methods are spread to his members, for example, if damage are dealt to a **Camp** all the damage will be dealt to its members.
This system allow the possibility of several **Ability** target mode like healing an entire camp for example.

Talking about **Abilities**, these a synchronized using a **Bolt Event** mainly containing the **Attribute ID, Ability ID, Launcher ID, Victim ID**, so when an ability is used all other clients can reproduce it once the event is received by getting the right **Attributes** in **Libraries** and using the ability.

Modifiers are effects which **remain for 1 turn** on its target, 2 classes inherit from it, **Buff** and **Debuff** which create an a tree with all others effects inheriting from these two.
It allows several method like removing a modifier inheriting from a certain type which ease the whole **Modifier** management.

Mobs AI is pretty basics, they will execute a **random attack** each time their **turn start** using a **ratio**. Depending on the **ability target mode**, they will execute their attacks on **their camp**, the **players camp**, **all camps**, **themselves**, or a **random target**. In the case of random target, we can specify if the mob should **target a member of his camp or not**, and if it should target the **lowest health target**.

Talking about **target mode**, the way it works is pretty simple to.
5 target modes are available :

- **Self** : the launcher itself
- **Mono** : another entity
- **Camp** : a camp
- **Opposite Camp** : the opposite camp of the targeted entity
- **Global** : everyone

Each **effects** has a target mode, these effects can be **primary** or **secondary**.
In an effect chain, the **first effect** target mode will be the **primary target mode** otherwise it will be **secondary**, it will adapt itself to the **primary target mode**.
For example, if the first effect target mode is **Mono** and the second effect target mode is **Opposite Camp**, it will **first** target **an entity** and **then** the **opposite camp of this entity**.

Workflow

In order to ease the whole game feedbacks implementation, a class **Game Events** was created. It's a scriptable object from which severals classes inherit, as **VFX Events**, **Camera Shake Events**, and so on. When an ability is used, it will in this idea execute all game events.

For animations, all characters are using **the same Animator,** the animations are swapped using the **Animator Override Controller** component.
In order to **ease character artist work**, all characters are animated using **2D IK** package, which allow to rig and deform sprites.

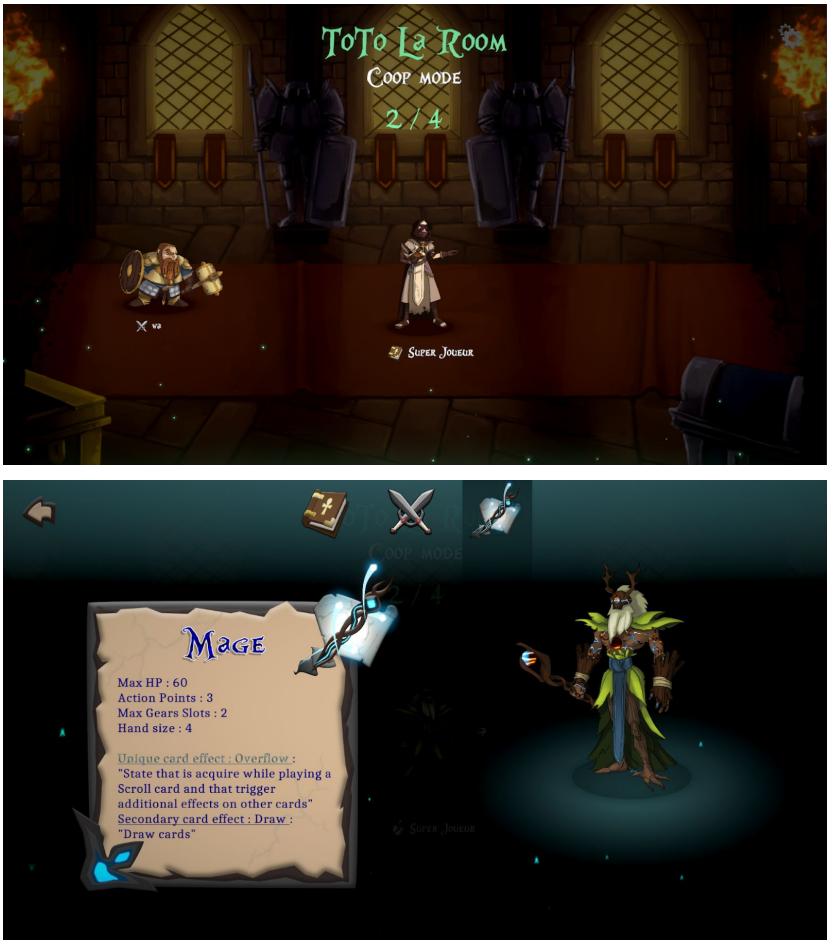
Networking

To access **Rooms**, players have to pass by a **Lobby** which display all the current rooms, with their name and player count.

When the join button is clicked, it open a panel which let the player **choose his name**, and confirm the join the **Room**. In the case the **Room** is full or start the game, it will **deny the access.**
It's also possible to create and room and choose the room name.

All of this is synchronized using classes inheriting from **Bolt Protocol Token** being a class with a **Read** and **Write** method and some method to convert basic value variable like int, float etc…
It can be used as parameters in several **Bolt** methods like for example on a player connection or using the **Bolt Instantiate** method.

Once in the **Room**, players will see all players and will have the opportunity to change their character, and finally start the game.



Map is **procedurally generated** by the **server** and instantiated using **Protocol Tokens** and extensions to synchronize each branch points composed of an int being the **point type** and a vector3 being the **position of the point**.

Once instantiated on the clients, a **Bolt Event** is sent to draw the line renderer between all points using bezier curves.

For the **Campfire** all cards contains a specific **Game Event** which trigger different methods. Once used, the card will send a **Bolt Event** containing the game event store it on the **server** which will execute all of these simultaneously on the morning.

During **Loot phases**, it's first click first pick, which means **realtime**, in order to avoid any kind of bugs, when a player click on a card, it will send a **Bolt Event** on the **server** in order to **ask the permission** to pick the card, if the card wasn't pick by someone already, the **server** will finally **authorize** the player to pick the card.

User Interface

User interface is mainly **managed** by a singleton called **Player UI**, which will handle all **persistent game UI** like the **banner** with the player name and different buttons, or the map for example.

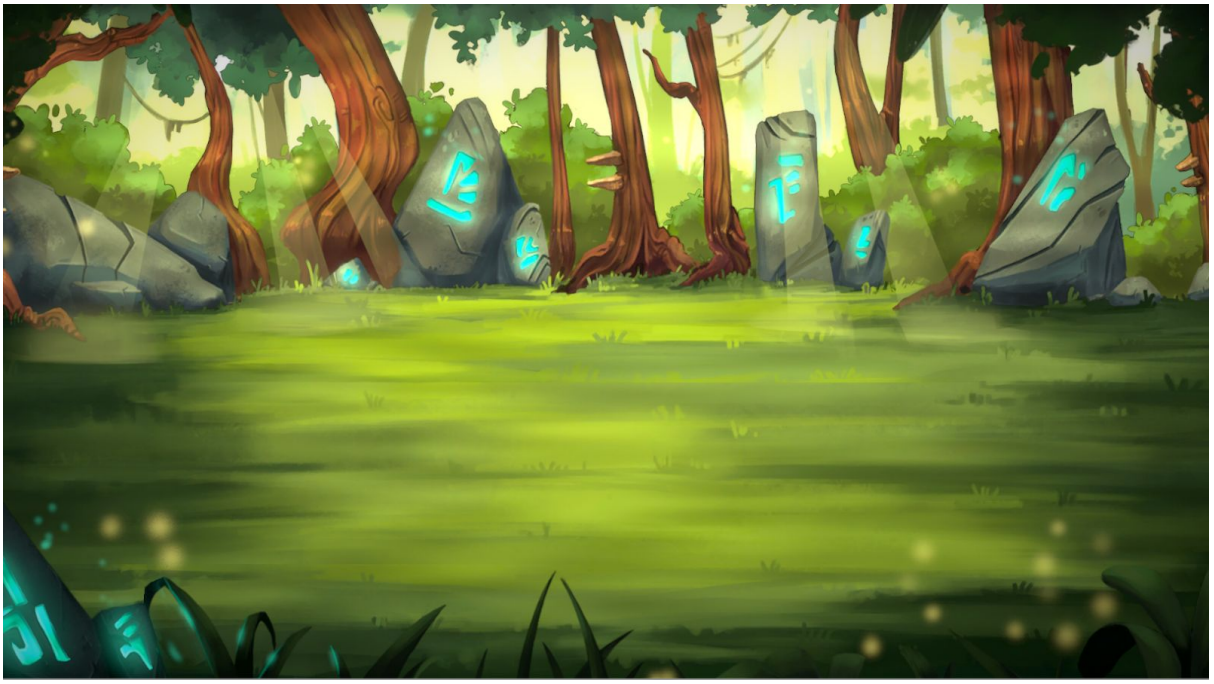
Some **network menu** were made to like the **lobby UI** or **rooms UI** using different Unity components like **Vertical/Horizontal Layouts** and **Content Size Fitters**.

Most menus are updating by adding their methods to **C# events** in order to only be their as **display** and not as value containers.

Graphics

Several shaders were made for **cards** to serve **two main purpose**, having **culling on card** so we can have a **recto verso effect**, and add an **appealing aspect** which enhance the principle of **quality** using **noise** and **mask**.

For the **forest environment** a shader applying a slight **panning noise** was made to fake the **effect of shadow** provided by leaves on the ground adding some **different tints of green**.
For the **lights**, a simple **additive** shader make the **alpha value varies** using **sin** * **time** and a **remap**.



Fire was made by **distorting the uv** with **noise** and adding a **panning mask**.

For the **mage character** a **mask** is applied to change **only the white part** of the character, then **panning texture** is added to give that **magic fluid aspect**. Finally a **panning noise** with the **opposite color** is added to give some **fake relief** and **magic aspect**.