Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

CARD ARENA ONLINE: Sistema multijugador servidor-cliente para un juego de Cartas del género *Digital Collective Card Game*.

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: André Escrich González

Tutor: Adolfo Muñoz García

[2020-2021]

Resumen

Card Arena Online es un proyecto de desarrollo de videojuego centrado en la interacción en línea entre dos jugadores a través de una conexión cliente-servidor.

El videojuego está gran mente inspirado en videojuegos como *Hearthstone*[1], *Magic: The Gathering Arena*[2] y especialmente *Legends of Runeterra*[3].

El género de videojuegos en el que este proyecto está incluido se denomina como "*Collective Card Game (CCG)*[4]" el cual se basa en la creación estratégica de mazos y enfrentamiento uno contra uno entre jugadores en línea.

El desarrollo de este proyecto es llevado a cabo únicamente por mí, André Escrich. Aunque la principal tarea llevada a cabo durante el desarrollo haya sido la programación, se han llevado a cabo diferentes tareas que consolidan lo que un videojuego necesita para ser creado. Entre ellas, modelado, creación de efectos especiales VFX y diseño.

Aparte de crear el videojuego, el principal objetivo del proyecto es crear un servidor que comunique a los diferentes clientes uniéndolos en partidas de dos y reenviando entre ellos las acciones del otro de manera sincronizada. Para ello, utilizaremos Unity3D como motor de juego para el desarrollo y usaremos las librerías de *sockets* de .NET Framework para la comunicación en línea.

Palabras clave: Videojuego, Multijugador, Cliente-Servidor, Unity3D, Cartas.

[1] Mas información <https://playhearthstone.com/en-us>
[2] Mas información <https://magic.wizards.com/en>
[3] Mas información <https://playruneterra.com/en-us/>
[4] Mas información <https://en.wikipedia.org/wiki/Collectible_card_game>

Abstract

Card Arena Online is a videogame development project focused on the online interaction between two players through a client-server-based connection

The videogame is highly inspired in videogame such as Hearthstone, Magic: The Gathering Arena and specially Legends of Runeterra.

The videogame genre in which this game is included is called “Collective card game (CCG)” which is based in the strategic deck creation and one versus one online duel.

The project development is done only by me, André Escrich. Even though the main task that had place during development was coding, there has been different instances of task that consolidate what a game needs to be created such as modelling, visual effects and designing.

In addiction of creation the videogame, the main objective of the project is creating a server that communicates the different clients matching them in games of two players and sending each other’s actions synchronously. To achieve this, we will use Unity3D as game engine for development and we will use the socket libraries from the .net framework for the online communication.

Keywords: Videogame, Multiplayer, Client-Server, Unity3D, Cards.

Tabla de contenidos

1 Introducción

1.1 Motivación

La principal motivación de este proyecto es la demostración de la capacidad de recreación de videojuegos de empresas multimillonarias como *Riot Games*[5] y su producto *Legends of Runeterra*. Este tipo de producto es llevado a cabo por desarrolladores y artistas de múltiples campos y entender como todas las piezas se complementan entre ellas es un aspecto esencial para desarrollar una versión similar por solo un individuo.

En el caso de una empresa *indie* podríamos encontrar como mínimo una separación de roles entre artista, programador y diseñadores siendo este último un puesto que se podría llevar a cabo incluso por uno de los dos primeros mencionados. A medida que una empresa va creciendo esta separación se va separando en diferentes tareas, programación se dividiría en *front-end* y *back-end*, siendo el programador *front-end* encargado de lo que el usuario ve y experimenta mediante el videojuego y el *back-end* el encargado de todas esas tareas que no son perceptibles a primera vista como creación de servidor, base de datos…

Sabiendo esto, se ha buscado un tipo de videojuego que ponga más carga al apartado de implementación de código que el apartado artístico, que, aunque existente, la cantidad de trabajo necesaria para tener un producto decente no es tan significativa como un producto que necesite animaciones de para modelos humanoides o grandes cantidades de paisajes. Por ello, se ha optado por un simple juego de cartas con un escenario estático para todas las partidas y cartas donde se ha usado ilustraciones de cartas regulares de póker para diferenciarlas.

Además de la recreación de un videojuego de cartas también se ha buscado que es lo que diferenciaría este proyecto de otros productos ya existentes en el mercado. Estos, ponen especial énfasis en las ilustraciones de las cartas siendo normalmente personajes fantásticos que intentan atraer a una audiencia joven. Teniendo esto en cuenta, y con

5 Mas información <https://en.wikipedia.org/wiki/Riot_Games>

el uso de ilustraciones que representen cartas comúnmente conocidas, este proyecto busca introducir una audiencia más adulta en este género de videojuegos.

1.2 Objetivo

Los principales objetivos son:

- Conseguir la creación de un cliente videojuego que pueda comunicarse con un servidor y comunique las acciones con otro cliente sin desincronizarse.

- Creación de un videojuego usando *Legends of Runeterra* como ejemplo a seguir para lograr una experiencia satisfactoria para potenciales usuarios.

- Poner en práctica diferentes conocimientos obtenidos durante la carrera como, por ejemplo, programación orientada a objetos de la asignatura “Ingeniería del software”, conocimientos sobre conexiones mediante el protocolo TCP estudiadas en asignaturas como “Sistemas y servicios en red” y asignaturas relacionadas con videojuegos como “Introducción a la programación de videojuegos” o *Animation and design of videogames*”.

- Creación de un videojuego dentro del género CCG que apunte a una audiencia más adulta.

- Crear un producto digno para mostrar en el portafolio técnico personal para futuras posibilidades de empleo en la industria.

1.3 Estructura

La estructura de esta memoria cuenta con tres partes. Una primera parte es un estudio sobre el género del videojuego en el que este proyecto se incluye, muestra de ejemplos de productos existentes dentro de la industria y la propuesta que este proyecto ofrece y comparte frente a ellos.

Una segunda parte, relacionada con el diseño, explica la división en bloques de cada sistema implementado, relaciones entre clases empleando diagramas UML y un recorrido a través de todas las tecnologías y herramientas utilizadas explicando el porqué de cada una de ellas.

Y una tercera parte, sobre el desarrollo de la solución, dividida en etapas, una correspondiente a la comunicación cliente-servidor, como ha sido implementado para este caso, mencionando únicamente el código esencial o complejo para el funcionamiento del sistema y visualizaciones de como el intercambio de información se lleva a cabo entre los clientes siempre teniendo en cuenta que estos estén sincronizados.

Como segunda etapa de la tercera parte, se explica la implementación del videojuego y las diferentes técnicas llevadas a cabo usando Unity3D[6], Blender[7] y Photoshop para conseguir la recreación del producto deseado. Se muestra código relacionado con el videojuego y el uso de polimorfismo y herramientas de programación para conseguir un código limpio y eficiente. Se mostrarán obstáculos que han ido surgiendo durante el desarrollo y las soluciones tomadas siempre con el objetivo de crear la mejor experiencia para los posibles jugadores.

---

[6] Software dedicado al desarrollo de videojuegos.
Mas información <https://en.wikipedia.org/wiki/Unity_(game_engine)>
[7] Software dedicado especialmente a modelaje 3D.
Mas información <https://es.wikipedia.org/wiki/Blender>

2 Contexto Tecnológico

Antes de poder hablar de los productos digitales, tenemos que hacer mención del producto a partir del cual estos se originaron; los juegos de cartas físicos.

La existencia de juegos con cartas lleva siendo conocida desde el siglo 9 en la dinastía China. Se puede mencionar las cartas de beisbol como precursor de lo que hoy en día conocemos como CCG, *Card Collective Game*.

Este género se consolidó en 1993 cuando Richard Garfield creó el juego *Magic: The Gathering* siendo el primer juego de cartas donde los jugadores no podían comprar todas las cartas de golpe, en vez de eso, jugadores comprarían una cantidad inicial de cartas y posteriormente comprarían paquetes de cartas para expandir sus mazos.

En la actualidad, la versión digital de CCG, DCCG, *Digital Card Collective Game*, están ganando más popularidad debido a su más fácil accesibilidad y la posibilidad de jugar contra oponentes de y desde cualquier lugar.

La popularidad de este género se debe a dos factores:

- La colección de cartas dándole al jugador una sensación de logro y avance. Permitiéndole la creación de barajas propias con estrategias personalizadas que, aunque puedan ser similares a las de otros jugadores, pueden ver sus decisiones reflejadas en las cartas elegidas.

- El desafío mental, las decisiones que el jugador toma y las diferentes combinaciones de cartas que se pueden llevar a cabo a lo largo de un duelo. Esto otorga al juego un nivel de competitividad adictivo que lleva a jugadores querer mejorar y aprender.

A continuación, se listarán productos similares al presentado en este TFG.

2.1 Hearthstone

Desarrollado por *Blizzard Entertainment*[8] en 2014 es posiblemente uno de los títulos más conocidos dentro del género CCG en el ámbito digital. Visualmente basado bajo la temática de personajes de *World of Warcraft*[9], ha conseguido llamar la atención de jugadores provenientes de esta franquicia. Con 23.5 millones de jugadores en 2020 sigue atrayendo nueva audiencia debido al constante número de actualizaciones añadiendo diverso contenido como nuevos mazos y modos de juego manteniendo el juego activo desde el lanzamiento y en boca de todos los interesados en el género.

Hearthstone se convirtió en un punto de referencia para futuros productos relacionados con la temática.

Ilustración 1: Imagen promocional de Hearthstone

[8] Publicadora de videojuegos estadounidense. Mas información: <https://en.wikipedia.org/wiki/Blizzard_Entertainment>
[9] Mas información: <https://en.wikipedia.org/wiki/World_of_Warcraft>

2.2 Magic: The Gathering Arena

Desarrollado y publicado por *Wizards of the Coast* en 2019 surge siendo la versión digital del juego de cartas físico *Magic: The Gathering* que lleva circulando desde 1993 creado por la misma compañía.

Tras el éxito como la entrega anteriormente mencionada *Hearthstone*, los creadores de Magic decidieron adentrarse en una versión digital siguiendo la misma jugabilidad.

Ilustración 2: Imagen promocional de Magic: The Gathering Arena.

2.3 Legends of Runeterra

Publicado el abril de 2020, la empresa conocida por *League of Legends*, *Riot Games*, decide desarrollar un videojuego de cartas inspirado en la versión física de *Magic: The Gathering* cambiando diferentes aspectos para hacerlo más accesible para nuevos jugadores.

Riot Games comienza así su expansión a diferentes géneros de videojuegos con esta entrega usando personajes y escenarios con temática relacionada con el videojuego que son conocidos por.

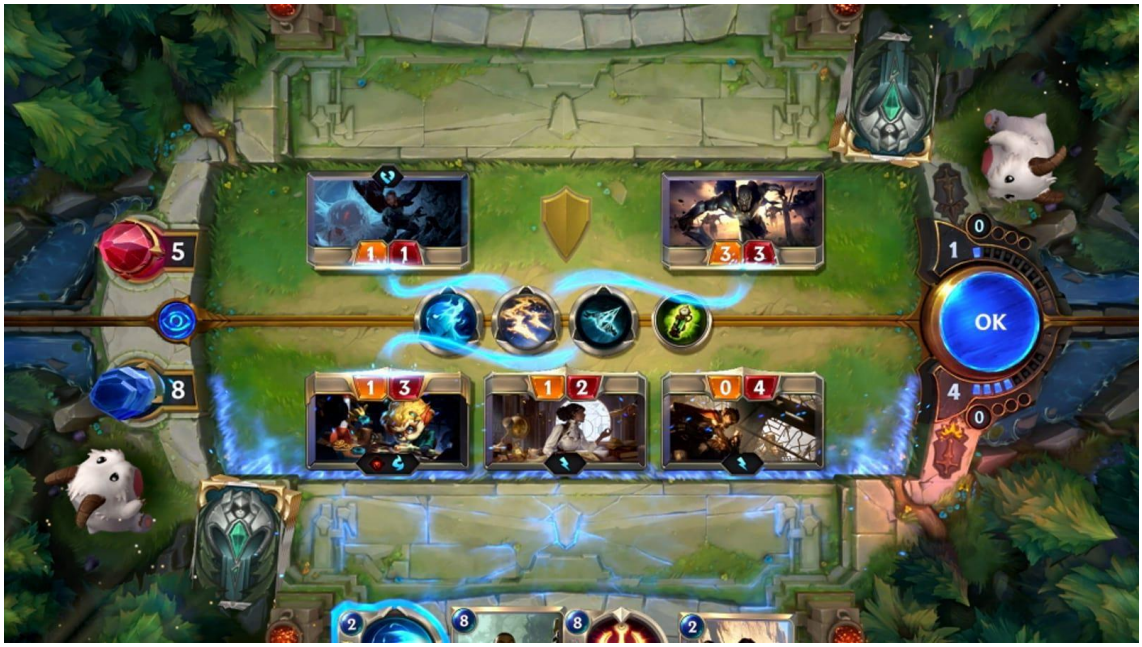


Ilustración 3: Imagen promocional de Legends of Runeterra

2.4 Propuesta

Viendo los ejemplos de productos mostrados en la sección anterior podemos observar grandes similitudes entre ellos, pero coinciden en el uso de ilustraciones orientadas a la fantasía y ciencia ficción.

Este proyecto, al igual que los ya existentes, tendrá una jugabilidad y composición similar, pero con una temática visual que intente ser más cómoda y reconocida por un público adulto que no esté interesado en fantasía.

3 Metodología

La metodología escogida para este desarrollo es el **modelo de cascada.**

3.1 Modelo cascada

El modelo de cascada consiste en una planificación linear dividiendo el desarrollo en fases que se llevarán a cabo secuencialmente donde cada fase es dependiente de la fase anterior.

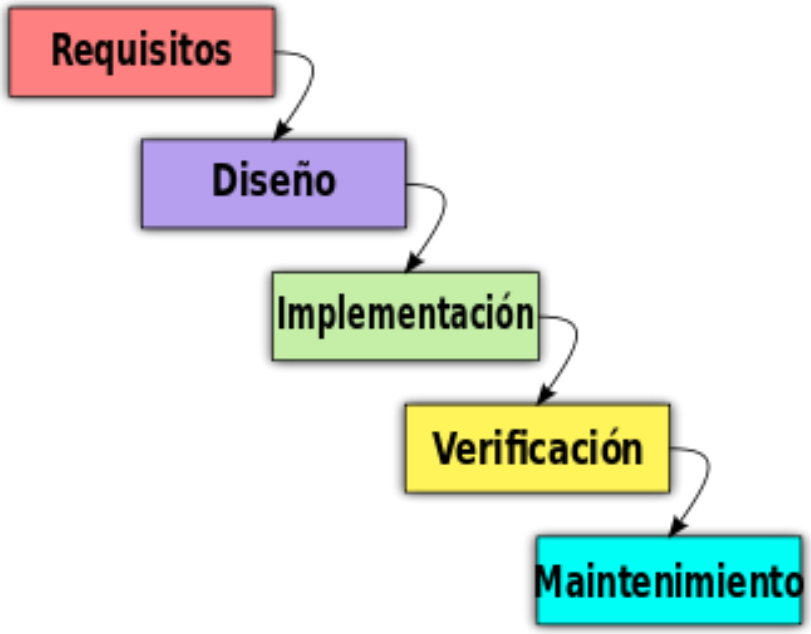


Ilustración 4: Modelo Cascada

3.2 Fases del modelo cascada

- **Requisitos**: Todos los posibles requisitos del sistema son recopilados y documentados en esta fase, aquí es donde crearemos nuestro documento de especificación de requisitos.

- **Diseño**: Los requisitos de la primera fase son estudiados y en esta fase se preparará el diseño del sistema. Aquí hablaremos de la arquitectura del sistema, hardware y tecnologías utilizadas.

- **Implementación**: Partiendo del diseño del sistema, desarrollaremos los diferentes módulos que componen nuestro proyecto.

- **Verificación**: Los módulos creados son integrados en el sistema y se hace una comprobación asegurando un comportamiento adecuado.

- **Mantenimiento**: Una vez el proceso de desarrollo esté completo, el proyecto pasará por diferentes pruebas y en el caso de la existencia de algún problema, mantenimiento será la fase donde se implementarán estos arreglos.

3.3 Ventajas e inconvenientes

Aunque este modelo destaque por su simpleza hay inconvenientes que le acompañan.

A continuación, se listarán algunas ventajas y razones por las que se ha usado este modelo y sus inconvenientes.

3.3.1 Ventajas

- **Simpleza**: el modelo cascada es un modelo muy sencillo de entender y usar, debido a ello, es uno de los modelos más usados en el desarrollo software.

- **Cantidad de documentación**: Equipos que usan este modelo requieren menos tiempo para documentación y pueden dedicar más tiempo al desarrollo.

- **No se necesita que el consumidor este envuelto durante el desarrollo**.

3.3.2 Inconvenientes

- **No hay cabida para modificaciones en mitad del desarrollo:** Los requisitos deben estar claramente definidos al inicio del desarrollo. Si el consumidor decide hacer cambios posteriormente, este modelo no da entrada a esas modificaciones.

- **No hay entregas intermedias:** El producto a entregar al consumidor es el existente tras todo el proceso de desarrollo.

- **Limitada interacción con el consumidor:** Aun listada como ventaja anteriormente, en diferentes escenarios puede considerarse un inconveniente.

3.4 Conclusiones

Debido a la naturaleza de este proyecto siendo este un trabajo de fin de grado se ha decidido optar por una metodología sencilla donde la existencia de un consumidor no es necesaria durante el desarrollo de este.

4 Especificación de requisitos

4.1 Introducción

Con el objetivo de definir los requisitos del sistema, existe un proceso de ingeniería llamado especificación de requisitos software que indica las funciones y restricciones que debe seguir el proyecto.

En este desarrollo seguiremos el estándar IEEE 830/1998 que define las secciones y apartados necesarios en un documento de especificación de requisitos.

4.1.1 Propósito

El propósito de esta sección es listar los requerimientos necesarios para implementar un videojuego de tipo *Collective card game* multijugador.

La persona a la que está sección estuviese dirigida sería un potencial inversor interesado en financiar este proyecto y al desarrollador de este.

4.1.2 Ámbito

El producto software a diseñar es denominado *Card Arena Online*.

Compuesto de un primer módulo cliente siendo la parte que los usuarios verán y podrán interactuar con y un segundo modulo servidor encargado de la comunicación entre usuarios.

Este desarrollo aspira a crear un producto dentro del campo del entretenimiento digital para un largo número de clientes que decidan invertir dinero en la compra de futuras actualizaciones.

4.1.3 Definiciones, Acrónimos y Abreviaturas.

CCG: *Collective Card Game*

DCCG: *Digital Collective Card Game*

IDE: *Integrated Development Environment*

TCP: *Transmision Control Protocol*

UDP: *User Datagram Protocol*

POO: Programación orientada a objetos

VFX: *Visual Effects*

4.1.4 Visión General de la Especificación de Requisitos.

La especificación de requisitos está dividida en tres secciones.

1. Introducción, conteniendo su propósito, ámbito, definiciones, acrónimos y abreviaturas usadas en esta memoria.
2. Descripción General, constituido por la perspectiva del producto, sus funciones, características de los usuarios, restricciones, suposiciones y dependencias.
3. Requisitos Específicos, compuesto de interfaces de usuarios, requisitos funcionales, requisitos no funcionales, análisis y validación.

4.2 Descripción General

4.2.1 Perspectiva del Producto

Este proyecto es independiente de cualquier otro sistema, creando así su propio servidor, por lo que se desarrollará teniendo en cuenta únicamente los requisitos de nuestro sistema.

4.2.2 Funciones del Producto.

La principal función del proyecto es el entretenimiento de los usuarios a través de una jugabilidad de enfrentamientos competitivos. Además, se busca la obtención de ingresos para el equipo de desarrollo mediante la creación de actualizaciones periódicas de contenido.

4.2.3 Características de los Usuarios

Los usuarios que usen este producto serán los jugadores.

No existen requisitos para el uso de este producto, pero existen características que definen a los usuarios que podrán entender y disfrutar más del producto:

- Personas experimentadas con juegos de cartas que quieran experimentar una nueva manera de jugar con ellas.
- Personas que hayan jugado con anterioridad a juegos DCCG similares.

4.2.4 Restricciones

- El proyecto será desarrollado usando Unity, C# y framework .NET y las limitaciones de sistema que conlleven.
- Es necesario disponer de conexión a internet en todo momento.
- La interfaz del juego estará influenciada por la actual competencia en el mercado.
- Se confiará que los usuarios no intenten aprovecharse usando herramientas de terceros para hacer trampas en el juego.
- El servidor estará instanciado en un dispositivo hardware limitado pero suficiente para las dimensiones correspondientes a un trabajo de fin de grado.

4.2.5 Suposiciones y Dependencias

Debido a su naturaleza de videojuego 3D es necesario que el dispositivo que lo ejecute cuente con suficiente potencia gráfica. Conexión a internet en el dispositivo es obligatoria en todo momento para la comunicación entre clientes. La jugabilidad se lleva a cabo únicamente a través de inputs de ratón, por lo que es necesario la disponibilidad de este.

4.3 Requisitos Específicos.

Antes de comenzar con el desarrollo del proyecto se llevó a cabo una etapa de investigación y debate con el tutor para tomar decisiones de que estilo de juego y referente se iba a usar. Decisiones sobre el apartado visual y artístico también se llevaron a cabo.

4.3.1 Interfaces de Usuario

Esta sección mostrará los bocetos de la interfaz del videojuego. Toda la interfaz se ha hecho a través de Unity.

La ilustración a continuación muestra la primera pantalla que ve el usuario al ejecutar el videojuego. Dándole la opción de jugar iniciando así la búsqueda de contrincante y la posibilidad de cerrar el videojuego. A la derecha mostrarían nuevas actualizaciones de contenidos.

Ilustración 5: Mockup de la interfaz del menú inicial

A continuación, se muestra la interfaz de la partida mostrando en ella la división del escenario en la parte del jugador y la parte del contrincante. Además, podemos observar elementos de jugabilidad esenciales como la posición donde los puntos de salud de cada jugador irán y el botón que permitirá controlar el flujo de la partida dándole a los jugadores diferentes opciones dependiendo de la situación.

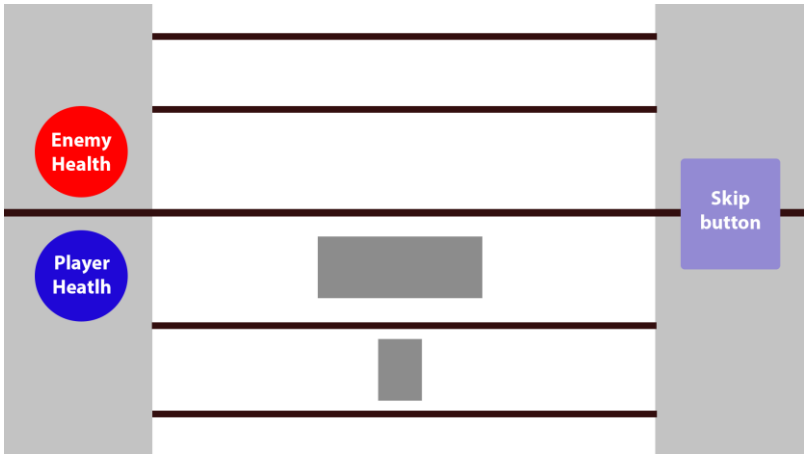


Ilustración 6: Mockup de la interfaz en partida

4.3.2 Requisitos Funcionales

A continuación, se listarán las acciones que se incluyen en nuestro videojuego.

- Buscar partida:
  - Objetivo: Permitir al jugador encontrar contrincante y empezar a jugar.
  - Entradas: El usuario selecciona “Play” en el menú inicial.
  - Proceso: Tras seleccionarlo, la acción se enviar al servidor y este buscará a otro jugador esperando y los juntará en una partida
  - Salida: Ambos jugadores ven como la segunda interfaz aparece y la partida comienza.

- Interactuar con carta:
  - Objetivo: Colocar la carta seleccionada en el lugar indicado.
  - Entradas: Jugador pulsa carta y la arrastra al lugar que el escoja y pueda.
  - Proceso: El juego calcula la posición inicial y a donde se mueve la carta y comprueba condiciones para verificar que la decisión del jugador es posible.
  - Salida: La carta se coloca donde corresponde y el juego reacciona según convenga. Ejemplo: Pasa turno automáticamente.

- Interactuar con habilidades de cartas.
  - Objetivo: Dependiendo de la carta, llevar a cabo su habilidad especial.

- Entradas: Una vez colocada la carta, esta pedirá una segunda elección siendo esta la carta objetivo de la habilidad.
- Proceso: El juego comprueba cual es la habilidad la ejecuta sobre la carta objetivo.
- Salida: Carta objetivo sufre algún efecto negativo o positivo. Ejemplo: Consigue dos de vida.

- Pasar turno:
  - Objetivo: Ceder el turno al contrincante.
  - Entradas: Jugador pulsa el botón “Skip Turn”
  - Proceso: Los inputs del jugador se bloquean y se desbloquean los inputs del contrincante para que pueda tomar decisiones como interactuar con cartas.
  - Salida: El contrincante puede tomar decisiones y actuar según el decida.

4.3.3 Requisitos no Funcionales

Los requisitos no funcionales indican requisitos relacionados con la calidad del proyecto y no están ligados a las acciones del usuario.

- En el caso de que haya más de un jugador buscando partida, el emparejamiento tiene que durar menos de cinco segundos.
- La sincronización entre los dos clientes sea idéntica el 99.99% de las situaciones.
- Que la partida no se bloquee porque ambos jugadores no tengan ninguna acción
- Que el juego siempre de información sobre lo que está pasando, a quien le pertenece el turno y las opciones que tiene el jugador.

4.3.4 Análisis

En esta sección se definirá el comportamiento de cada componente y la interacción entre ellos.

Para dar contexto a nuestro sistema usaremos Casos de Uso para visualizar las posibles acciones que se pueden llevar a cabo por nuestros usuarios y servidor y las consecuencias que conlleva cada una de ellas.

A continuación, se mostrará los Casos de uso de este proyecto e iremos especificando cada uno de ellos individualmente.

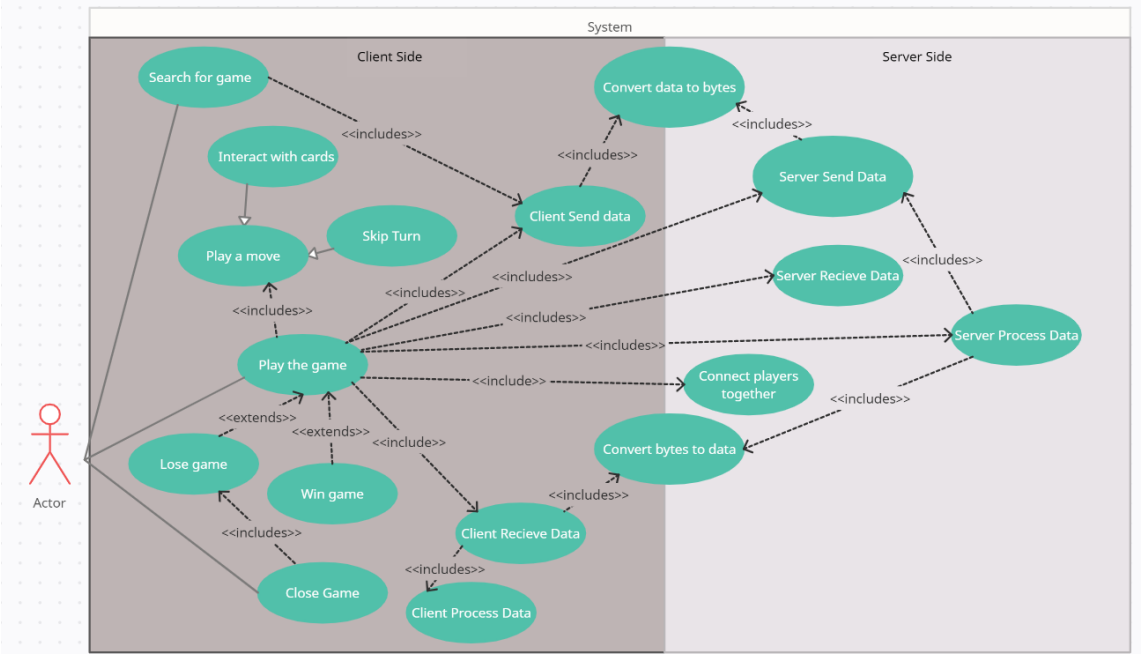


Ilustración 7: Caso de uso de nuestro sistema

Los usuarios externos que componen nuestro sistema son: un potencial jugador y nuestro modulo servidor

Caso de uso: Search for Game	
Resumen	Jugador entra en estado de espera hasta que el servidor empareje a dos jugadores para empezar una partida.
Actor	Usuario
Precondición	Que el usuario esté en el menú inicial.
Descripción	Pulsar el botón “Play”
Postcondición	El cliente entrará en estado de espera de partida.

Caso de uso: Close Game	
Resumen	Se cierra la aplicación
Actor	Usuario
Precondición	Que el usuario esté en el menú inicial.
Descripción	Pulsar el botón “Close”
Postcondición	La aplicación se cierra

Caso de uso: Play the game	
Resumen	Jugador participa en la partida.
Actor	Usuario
Precondición	Que se haya establecido conexión con otro jugador.
Descripción	Jugador interactúa con el escenario y cartas.
Postcondición	El cliente se actualiza en función de las acciones del jugador y se prepara para enviar la correspondiente información al servidor.

Caso de uso: Win Game	
Resumen	La partida finaliza y el jugador gana la partida.
Actor	Usuario
Precondición	Que el jugador esté en partida y que consiga bajar la salud del contrincante al cero.
Descripción	Colocar cartas para que ataquen al contrincante.
Postcondición	El mensaje de “Victoria” aparece y el usuario es enviado al menú inicial

Caso de uso: Lose Game	
Resumen	La partida finaliza y el jugador pierde la partida.
Actor	Usuario
Precondición	Que el jugador esté en partida y que el contrincante le baje la vida a cero.
Descripción	No colocar cartas para defenderse de las cartas del oponente
Postcondición	El mensaje de “Derrota” aparece y el usuario es enviado al menú inicial

Caso de uso: Interact with cards	
Resumen	El jugador interactúa con cartas y el cliente reacciona correspondientemente
Actor	Usuario
Precondición	Que el jugador esté en partida y que sea su turno.
Descripción	Flujo: 1. Pulsar una carta disponible 2. Mover el ratón a un posible destino 3. Pulsar para aceptar elección.
Postcondición	La carta se coloca en la parte del tablero escogida y cambia de aspecto según la situación. El turno se cede al contrincante. Información sobre la jugada se prepara para ser enviada al servidor.

Caso de uso: Skip Turn	
Resumen	El jugador le cede el turno al contrincante.
Actor	Usuario
Precondición	Que el jugador esté en partida y que sea su turno.
Descripción	Pulsar botón “Skip Turn”
Postcondición	El juego se actualiza cediendo el turno al contrincante y la información relacionada con la acción se prepara para ser enviada al servidor.

Caso de uso: Client Send Data	
Resumen	La información para enviar se empaqueta y es enviada al servidor
Actor	Usuario
Precondición	Que se haya establecido conexión entre dos jugadores.
Descripción	Realizar alguna jugada.
Postcondición	Servidor recibe información.

Caso de uso: Convert data to bytes	
Resumen	Los datos para enviar se transforman en bytes.
Actor	Usuario
Precondición	Que haya información que necesite ser enviada.
Descripción	Realizar alguna jugada.
Postcondición	La información se transforma a bytes y se preparan para enviarse.

Caso de uso: Server Receive Data	
Resumen	Se obtienen datos pasados a través de TCP
Actor	Usuario
Precondición	Datos se han enviado desde un cliente.
Descripción	Jugador realiza jugada
Postcondición	Los datos llegan y se preparan para ser procesados por el servidor.

Caso de uso: Client Receive Data	
Resumen	Se obtienen datos pasados a través de TCP
Actor	Usuario
Precondición	Que información haya sido procesada y enviada por el servidor
Descripción	Jugador realiza jugada
Postcondición	Los datos llegan y se preparan para ser procesados por el cliente.

Caso de uso: Connect players together	
Resumen	Servidor empareja a dos jugadores para empezar una partida.
Actor	Usuario
Precondición	Que ambos usuarios estén en el menú inicial.
Descripción	Flujo: 1. Jugador 1 pulsa “Play” y espera por otro jugador 2. Jugador 2 pulsa “Play”
Postcondición	Ambos clientes de los jugadores comienzan la partida y se establece una conexión entre los usuarios y servidor para enviarse datos.

Caso de uso: Server Send Data	
Resumen	Servidor envía información a un cliente.
Actor	Usuario
Precondición	Que los dos jugadores estén conectados en una partida.
Descripción	Flujo: 1. Uno de los dos jugadores realiza alguna acción 2. Datos son enviados a través de TCP al servidor 3. Los datos son procesados por el servidor.
Postcondición	Los datos llegan al usuario que no realizó la acción.

Caso de uso: Convert bytes to data	
Resumen	Los datos recibidos se transforman en datos útiles.
Actor	Usuario
Precondición	Que dos jugadores estén conectados y se envíen datos.
Descripción	Flujo a: 1. Información es recibida en servidor 2. Servidor transforma bytes a datos Flujo b: 1. Información es recibida en cliente 2. Cliente del usuario transforma bytes a datos
Postcondición	Información es traducida en alguno de los dos extremos y se prepara para procesarse.

Caso de uso: Server Process Data	
Resumen	Los datos útiles se evalúan para ser reenviados
Actor	Usuario
Precondición	Que haya conexión entre dos jugadores y se envíen datos.
Descripción	Jugador realiza acción
Postcondición	La información procesada se prepara para ser empaquetada y enviada al cliente opuesto.

Caso de uso: Client Process Data	
Resumen	Los datos útiles se evalúan para actualizar el cliente.
Actor	Usuario
Precondición	Que el cliente oponente envíe información
Descripción	Oponente jugador realiza acción
Postcondición	El cliente se actualiza dependiendo de los cálculos realizados con la información obtenida.

4.4 Validación

Para la validación de este desarrollo, se le expuso al tutor los requerimientos que constituyen al proyecto. En estas reuniones se tomaron decisiones sobre la jugabilidad, el comportamiento entre los clientes y apartados artísticos para consolidar las piezas fundamentales para crear el videojuego.

Tras llegar a una idea de proyecto que satisficiese tanto al alumno como al tutor se comenzó a hacer su diseño y desarrollo.

5 Diseño de la Solución

5.1 Arquitectura del Sistema

El proyecto está dividido en dos subsistemas:

- Servidor centralizado encargado de la comunicación entre un número indefinido de clientes.
- Cliente que contendrá la lógica y visualización del videojuego que hará conexión con el servidor para comunicarse con otro cliente.

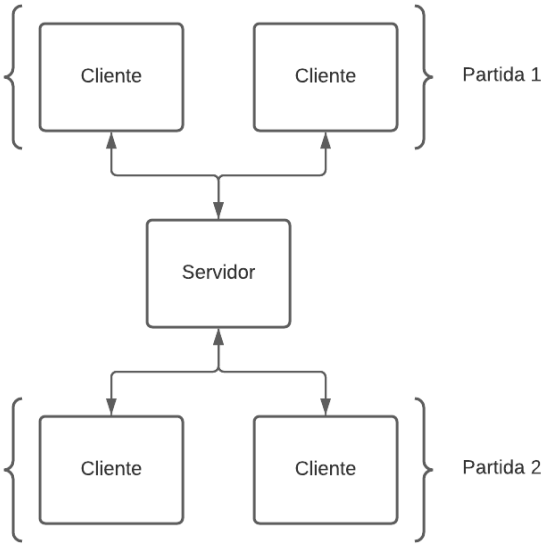


Ilustración 8: Diagrama de bloques relación Clientes-Servidor

Ambas implementaciones, tanto la del servidor y cliente serán desarrolladas en Unity3D. Unity permite hacer *Server builds* que convierten la solución en una consola de comandos lo cual es suficiente para lo que necesitamos para esta parte del proyecto.

El servidor llevará un recuento de clientes conectados y los irá emparejando según nuevos clientes conecten en partidas de dos jugadores. Las partidas son instancias aisladas, por lo que no tendrán ningún efecto sobre otras partidas. En conclusión, el servidor es el intermediario entre clientes y escucha desde que cliente viene la información y a que cliente reenviarla.

Una vez una partida entre dos jugadores es creada, el servidor no tendrá ninguna autoridad sobre la lógica de la partida. Su única funcionalidad será comunicar las acciones que el otro jugador haya hecho por lo que es importante que cada cliente escuche del servidor las acciones del otro cliente para mantener sincronizada la misma partida en ambas instancias.

La decisión de usar una arquitectura Cliente-Servidor en vez de alternativas como *Peer-to-Peer*[10] es la posibilidad de añadir al servidor la capacidad para procesar lógica y asegurar que ambos clientes estén sincronizados y evitar que uno de ellos haga trampas modificando con herramientas externas y desincronizar la información de ambos clientes como por ejemplo modificando el daño de una carta.

Debido a que no se quería aumentar la carga de trabajo, el servidor no hace ninguna comprobación de sincronización entre los servidores y confía que no se usen trampas.

Diseño detallado

En esta parte de la memoria se mostrará la distribución de las clases que componen el servidor y el cliente y la relación entre ellas.

Al ser dos instancias diferentes, comenzaremos con la parte del proyecto con menor cantidad de clases, el servidor.

[10] Modelo de comunicación entre dispositivos descentralizado.
Mas información <https://es.wikipedia.org/wiki/Peer-to-peer>

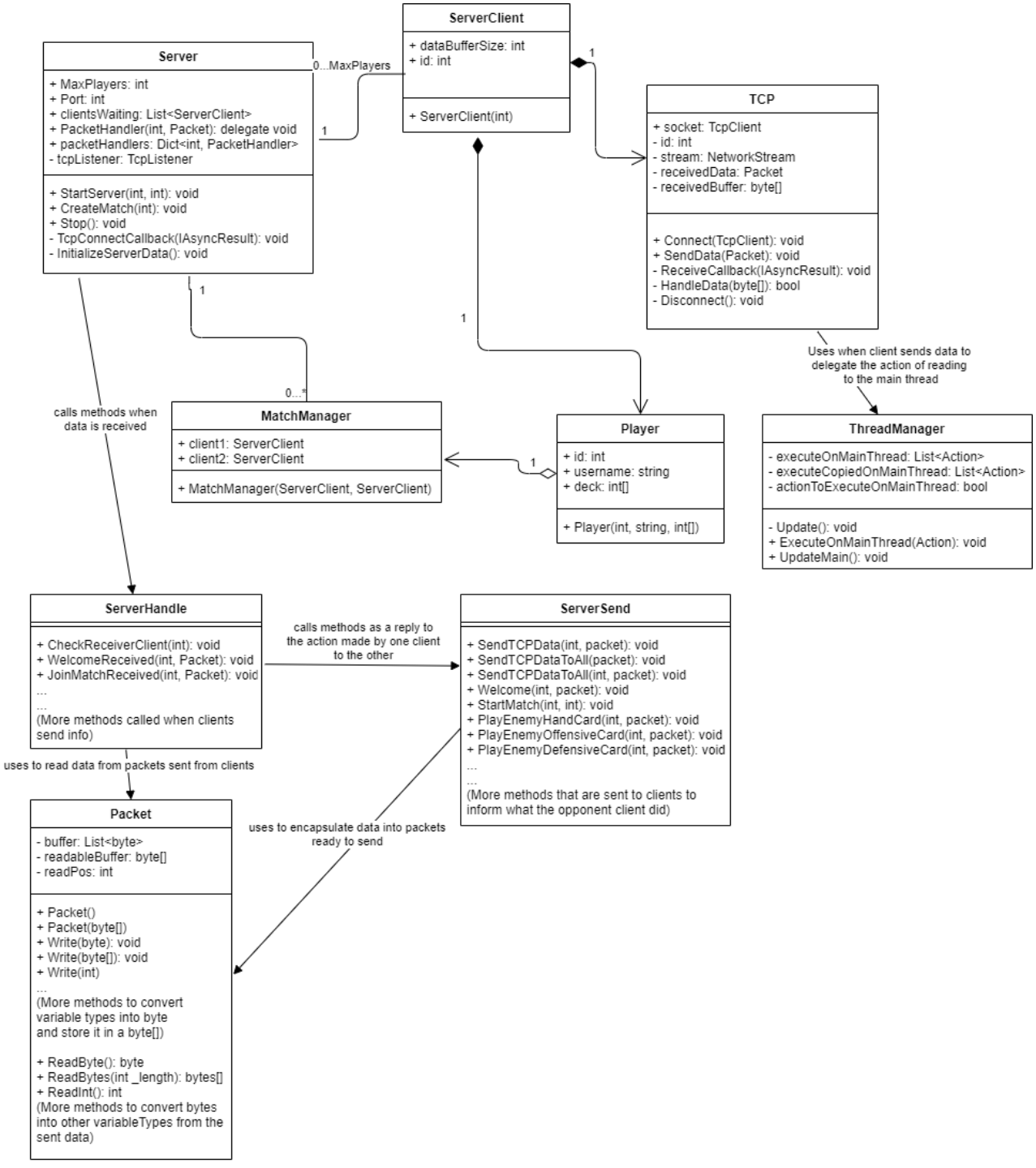


Ilustración 9: Diagrama de clases de la solución del servidor

Como podemos ver en el diagrama de clases anterior, el servidor consta de nueve clases imprescindibles para recibir información de los clientes, emparejarlos en partidas y reenviar información entre ellos cada vez que hacen una acción.

La solución del cliente contará con una parte similar a la del servidor encargada de la recepción y envío de paquetes y otra parte que constará de las clases que consolidan lo que es el videojuego en sí.

Debido al gran número de clases y a la extensión de la solución del cliente, dividiremos la visualización en dos diagramas, uno relacionado con la implementación relacionada con la comunicación con el servidor y otro diagrama para a lógica del videojuego.

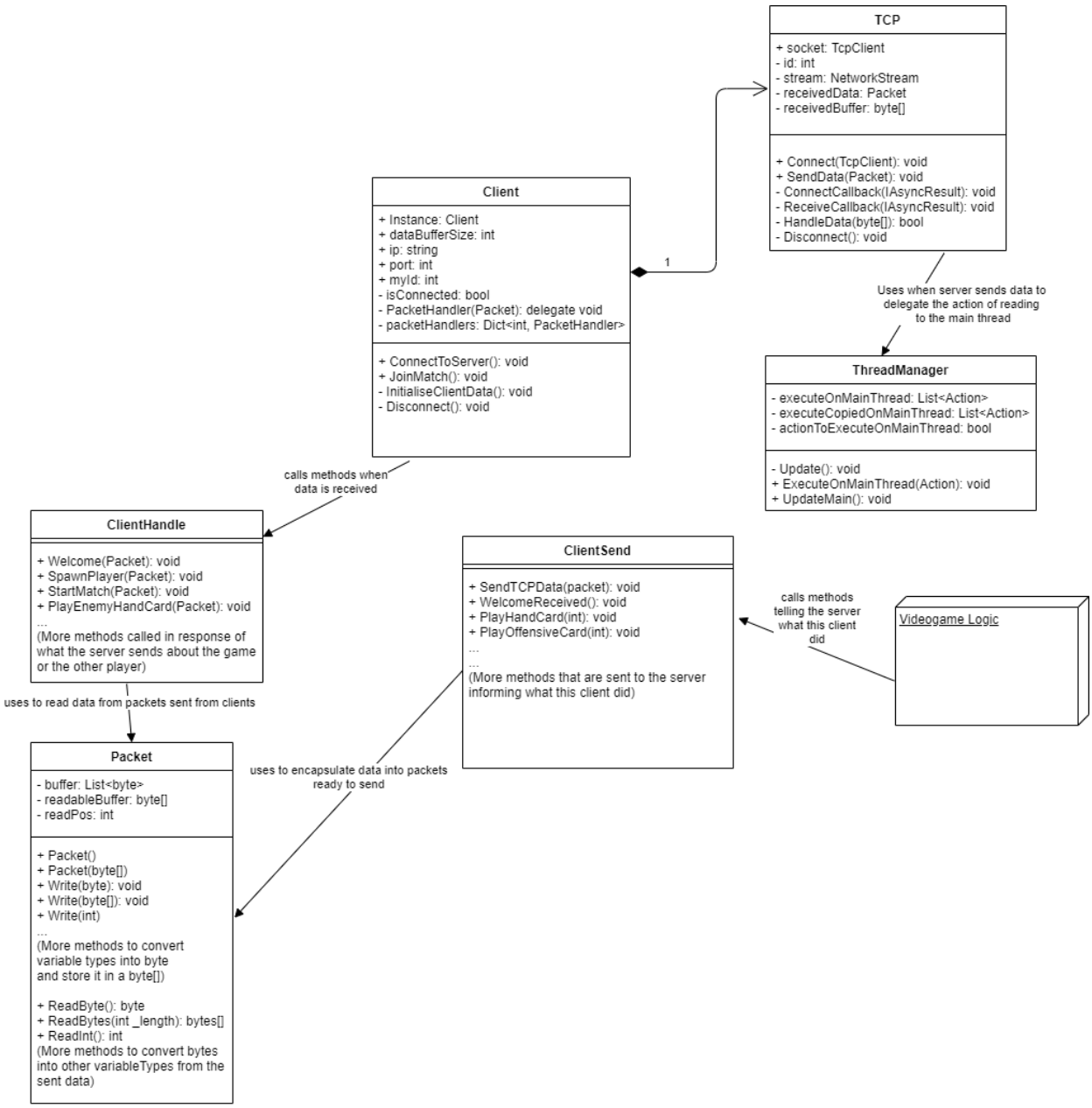


Ilustración 10: Diagrama de clases de la solución del cliente relaconado con la comunicación con el servidor

Como podemos observar, si comparamos el servidor y la parte de la cliente relacionada con la comunicación *online* podemos ver semejanzas como la existencia de clases "*Handle*" y "*Send*" relacionadas con métodos llamadas cuando información es recibida y enviada. También podemos encontrar el uso de la clase *Packet* encargada de encapsulación de información en byte *arrays* en ambas soluciones.

A continuación, se mostrará un diagrama mostrando la relación de clases del bloque "*Videogame Logic*" mostrado en el diagrama anterior ilustrando así la relación lógica detrás del videojuego de cartas.

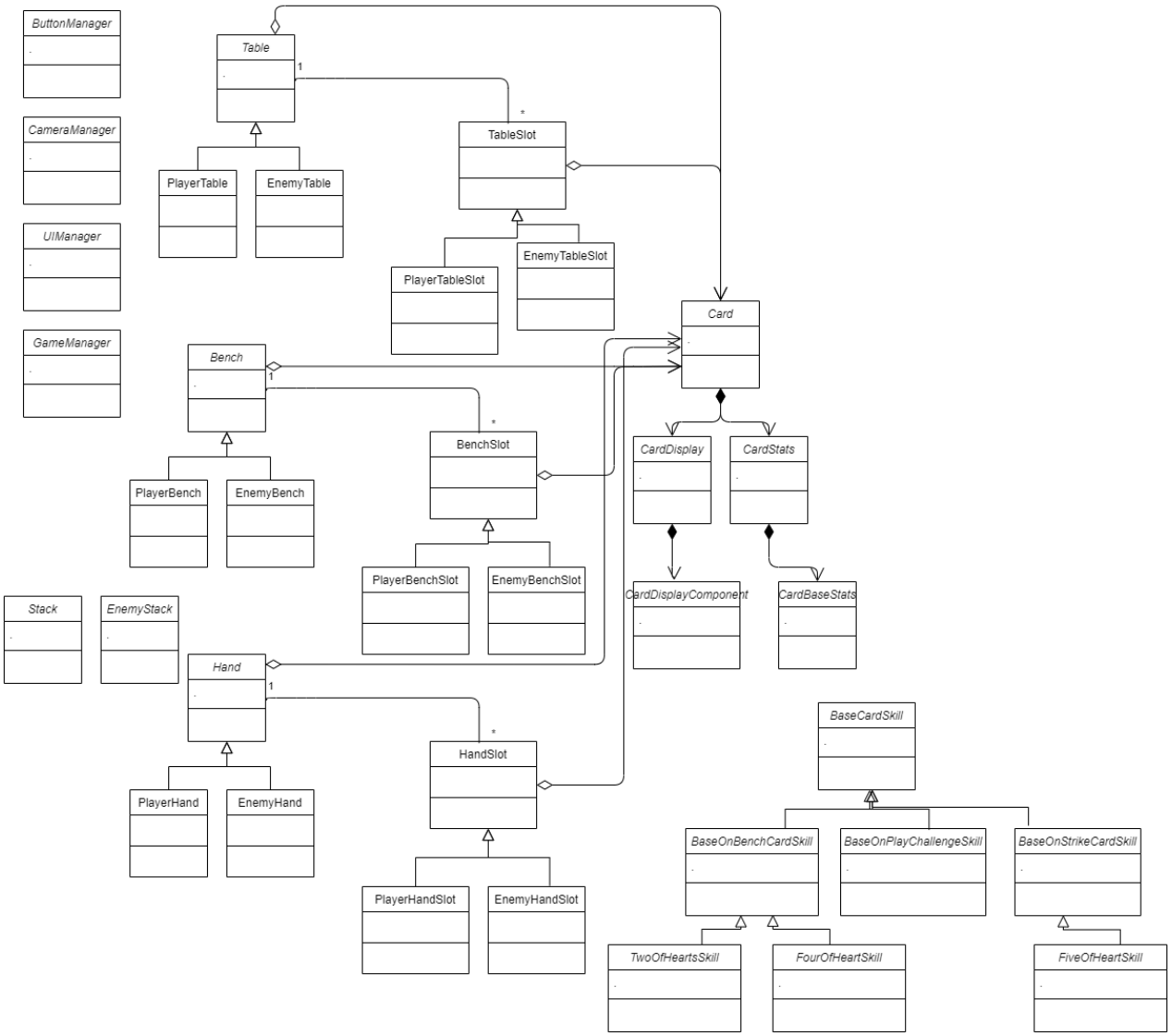


Ilustración 11: Diagrama de clases de la solución del Cliente relacionado con la implementación del videojuego.

En este caso, se ha decidido no añadir los atributos y los métodos para una más clara visualización de relaciones entre clases.

Como se puede observar, se ha implementado polimorfismo en casos como "*Table*", siendo esta la clase padre de "*EnemyTable*" y "*PlayerTable*" para evitar repetir implementaciones de código que son comunes entre las dos clases debido a su similar naturaleza. Esto último también puede verse representado en las diferentes habilidades de cartas que heredan de "*BaseCardSkill*".

5.2 Tecnología Utilizada

5.2.1 Motor gráfico:

Para crear un videojuego es necesario un programa que esté especializado en esta tarea. Las funcionalidades necesarias para estos suelen ser un motor de renderizado para gráficos 2D o 3D, físicas, detección de colisiones…

Grandes empresas cuentan con desarrolladores destinados únicamente a la creación de estos softwares como, por ejemplo:

- *Frostbite* creado por *Electronic Arts*
- IW creado por *Infinity Ward* y *Treyarch*
- *Source* creado por *Valve*.

Existe una gran cantidad de motores gráficos privados que empresas crean, pero se pueden encontrar otros destinados para uso público como pueden ser Godot, RPGMaker y mayormente conocidos, Unreal Engine 4 y Unity3D.

En mi caso, debido a mi mayor experiencia y cantidad de recursos didácticos en internet de cómo usarlo, he escogido Unity3D.

5.2.2 Lenguaje de programación:

En Unity3D, el scripting está hecho en C#, por lo que será el lenguaje que usaremos durante el desarrollo de este proyecto.

5.2.3 Entorno de desarrollo integrado:

Una IDE[11] es una herramienta esencial para facilitar la programación y desarrollo de cualquier aplicación. Proporcionado al desarrollador ayudas como autocompletación, compilación, *debugging*[12] y sugerencias para un código mejor estructurado.

Necesitaremos una IDE que tenga soporte para .NET. Para este proyecto he usado JetBrains Rider[13] con su licencia gratuita para estudiantes.

5.2.4 Control de Versiones:

Se ha usado Git para la creación de *backups* y mantener un registro de los cambios hechos durante el desarrollo. Permite la posibilidad de retroceder a versiones previas en caso de que no se pueda encontrar la solución de algún error en el programa. También nos facilita la posibilidad de transferir el proyecto entre diferentes dispositivos. Git también es una gran herramienta para facilitar la gestión de desarrollo entre varios miembros de equipo mediante la creación de *branches*[14] y fusiones de código, pero en nuestro caso, no necesitaremos usar estas herramientas.

Se ha utilizado GitKraken[15] y Fork[16] como aplicaciones que facilitan una interfaz para una sencilla y clara visualización de la información que Git nos ofrece.

5.2.5 Herramientas artísticas

Como se ha mencionado en la introducción, un videojuego es consolidado por múltiples aspectos y entre ellos el apartado artístico tiene gran peso por lo que se han precisado herramientas que nos ayuden a obtener estos objetivos.

Para la creación de modelajes y alteración de UV maps[17] para la futura aplicación de texturas sobre ellos, se ha utilizado Blender. Existen alternativas como Maya[18], pero debido a la previa experiencia con Blender, se ha escogido este para este proyecto.

---

[11] Aplicación que facilita el proceso de programación. Mas información <https://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado>
[12] Proceso de búsqueda y resolución de errores en aplicaciones o código.
[13] IDE .NET multiplataforma. Mas información <https://www.jetbrains.com/rider/>
[14] Iteración aislada del desarrollo en un proyecto que no afecta a otras iteraciones.
[15] Interfaz gráfica para Git. Mas información <https://www.gitkraken.com/>
[16] Otra Interfaz gráfica para Git. Mas información <https://git-fork.com/>
[17] Proceso en el modelaje 3D por el cual se procesa una imagen 2D por encima de la superficie 3D de un modelo.
[18] Software para modelaje, animación, simulación y renderizado.
Mas información en <https://www.autodesk.com/products/maya/overview>

Para todo lo relacionado con la creación de *assets*[19] en 2D y edición de *sprites*[20], se ha escogido Adobe Photoshop 2020.

A la hora de creación de efectos visuales, se ha utilizado Unity *ShaderGraph*[21] para la creación de materiales efectistas para animar de forma atractiva diferentes componentes en la escena.

5.2.6 Protocolo de capa de transporte[22]

Debido a que el nuestro videojuego es en línea y es necesario el transporte de información entre dispositivos para la sincronización de estos, es necesario decidir qué protocolo es óptimo para este caso en concreto.

Los posibles protocolos disponibles para la recepción y transmisión de paquetes con información son los siguientes:

- Protocolo de control de transmisión (TCP)
- Protocolo de datagramas de usuario (UDP)

Debido a que TCP confirma que los paquetes lleguen a su destino estableciendo una conexión punto a punto entre los hosts de envío y recepción, lo hace un protocolo fiable a diferencia de UDP que se usaría para envíos pequeños de datos que no supondría un gran problema que se perdiese.

Sabiendo esto, en el ámbito de videojuegos, TCP se usa para transportar información imprescindible como, por ejemplo, detección de colisiones entre clientes. UDP en cambio, se usaría para informar del cambio constante de posición de un jugador.

Para nuestro proyecto, teniendo en cuenta que la sincronización es imprescindible y que el número de paquetes no será muy elevado, se ha usado únicamente el protocolo TCP para casos como, por ejemplo, informar cuando un jugador se ha conectado o que carta y donde la ha colocado cada jugador.

---

[19] En el ámbito de videojuegos, es cada uno de los elementos que componen un juego.
[20] En el ámbito de videojuegos, es un conjunto de imágenes que representan un objeto de manera gráfica.
[21] Herramienta de Unity basada en nodos para el diseño de shaders y efectos.
Mas información <https://unity.com/shader-graph>
[22] Protocolos que permiten el transporte en línea de paquetes con información
Mas información <https://es.wikipedia.org/wiki/Capa_de_transporte>

6 Desarrollo de la solución propuesta

Para la explicación del desarrollo, dividiremos este apartado en una primera parte con la explicación del funcionamiento de la conexión e intercambio de paquetes y una segunda parte explicando los diferentes componentes que consolidan el videojuego.

Se mostrará el código únicamente relacionado con partes complejas o imprescindibles para el funcionamiento de los sistemas y se mencionará las etapas por las que problemas que han ido surgiendo han pasado hasta llegar a la solución que se considera óptima.

6.1 Primera Etapa: Desarrollo del conexión e intercambio de información.

6.1.1 Estudio de posibilidades para la creación del servidor

Siendo este componente la pieza esencial en la que está basado nuestro proyecto, se ha puesto especial interés en su correcta implementación y funcionamiento.

Antes de empezar a escribir cualquier línea de código se ha hecho un estudio de las posibilidades que se tienen sabiendo que se va a usar Unity3D.

Unity disponía de una herramienta multijugador propia llamada UNet[23] pero fue descontinuada y reemplazada por MLAPI[24], la cual se encuentra aún en desarrollo. Por lo tanto, se ha encontrado un buen reto en la búsqueda de una alternativa externa al motor gráfico que se encargue de la comunicación entre sistemas.

Existen alternativas de terceros conocidas como *Photon Bolt*[25] que se encargan de estos procesos, pero conllevan en uso y estudio de librerías que podrían ser desarrolladas con conocimientos aprendidos durante la carrera.

Tras un estudio de diferentes herramientas se llegó a la conclusión de la posibilidad de la creación de un sistema servidor-cliente propio usando librerías de sockets integradas en el *Framework .NET* que ya se usan para el desarrollo de *scripts* en Unity.

[23] Herramienta descontinuada de Unity <https://docs.unity3d.com/Manual/UNet.html>
[24] Herramienta en desarrollo de Unity para sistemas multijugador <https://docs-multiplayer.unity3d.com/>
[25] Mas información <https://www.photonengine.com/bolt>

6.1.2 Implementación de la conexión entre clientes y el servidor

Sabiendo esto, creamos un proyecto Unity, en nuestro caso, versión 2021.1.1f1, y creamos un *GameObject*[26] que funcionará como contenedor para los scripts que crearemos relacionados con la inicialización del servidor.

Creamos un script Server que funcionará como epicentro de la implementación y le importamos las librerías:

- System.Net
- System.Net.Sockets

Proporcionando así a nuestra solución todas las herramientas necesarias para comenzar a crear nuestro servidor.

Habiendo decidido con anterioridad el uso del protocolo TCP, hacemos uso de la clase *TcpListener* e implementamos lo necesario para la inicialización de este componente que escuchará la información proveniente de cualquier IP a través del puerto especificado.

Cada vez que nuestro *TcpListener* escuche nueva información proveniente de otro dispositivo, nuestro código se encargará de crear un nuevo objeto *ServerClient* creado por nosotros que guardará información del Cliente emisor, en nuestro caso, un nuevo jugador conectándose al sistema. A este cliente, a la hora de instanciación, se le proporciona un socket que usará como medio de comunicación con el servidor hasta su desconexión.

Con esto, ya tendremos una representación de cada jugador en la parte del servidor que a partir de ahora serán los encargados de escuchar el flujo de información y procesarla en caso de que sean ellos los que manden la información al servidor. También serán los encargados de enviar a sus respectivos clientes información que el servidor quiera comunicar.

A continuación, se muestra una representación gráfica de lo anteriormente implementado

[26] Componente base para todas las entidades en una escena de Unity.

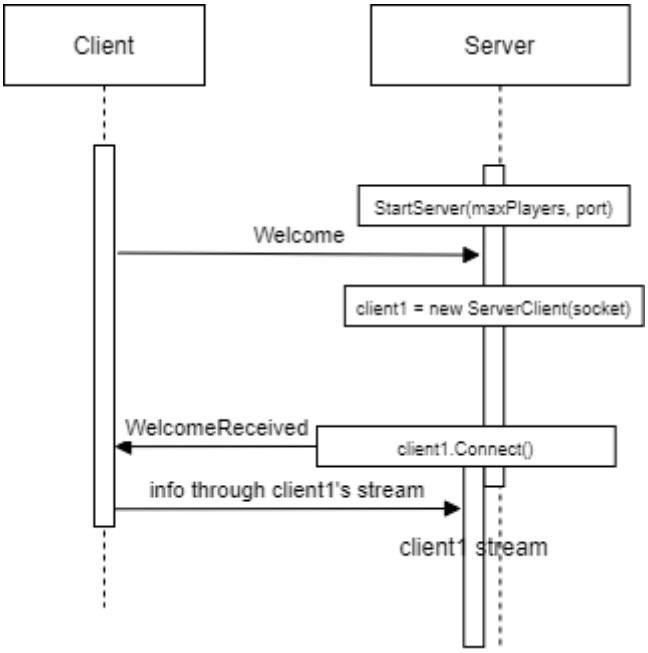


Ilustración 12: Comunicación entre cliente y servidor

Una vez creado el *ServerClient*, éste escuchará la posible información para posteriormente procesarla.

A continuación, se hará una breve explicación de las piezas que nos permiten procesar información y que hacer una vez el servidor sabe que nos está enviando cada cliente.

6.1.3 Empaquetado y traducción de variables

La información recibida y enviada serán tipos de variables primitivos, específicamente:

- *Short*
- *Int*
- *Long*
- *Float*
- *Bool*
- *String*

Y *arrays* de las mismas.

Estas variables serán transformadas en *arrays* de bytes para la envío y viceversa para la recepción e interpretación.

Este proceso lo vamos a llamar a partir de ahora “empaquetado” y todo lo relacionado a este viene implementado en la clase *Packet* que hereda de la interfaz *IDisposable*[27].

Cuando queramos enviar información a otro dispositivo, crearemos un objeto *Packet* al cual le pasaremos las variables que queremos como información a través de sus métodos para que el mismo objeto lo traduzca en información preparada para ser enviada mediante TCP.

A continuación, se mostrarán ejemplos de cómo la clase *Packet* traduce variables a bytes y bytes a variables:

```
public void Write(int _value)
{
    buffer.AddRange( collection: BitConverter.GetBytes(_value));
}
```

Ilustración 13: Método de conversión int a bytes

Como podemos observar, *Packet.Write(int)* recibe un valor *int* indicado como parámetro y lo añade a la *List<byte> buffer* de este *packet*.

Este *packet* con estos *bytes* serán los que posteriormente serán enviados mediante TCP y escuchados por el destinatario y traducidos usando un método similar como el que se muestra a continuación:

[27] Interfaz de .NET que proporciona un mecanismo para liberar recursos de memoria una vez utilizados. Más información <https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=net-5.0>

```
19 usages
public int ReadInt(bool _moveReadPos = true)
{
    if (buffer.Count > readPos)
    {
        // If there are unread bytes
        int _value = BitConverter.ToInt32(readableBuffer, readPos); // Convert the bytes to an int
        if (_moveReadPos)
        {
            // If _moveReadPos is true
            readPos += 4; // Increase readPos by 4
        }
        return _value; // Return the int
    }
    else
    {
        throw new Exception( message: "Could not read value of type 'int'!");
    }
}
```

Ilustración 14: Método de conversión bytes a int

Sabiendo que cada *int* está compuesto por 4 *bytes*, si leemos la lista de bytes obtenida y leemos *bytes* de 4 en 4, podemos traducir usando *BitConverter.ToInt32* para obtener el *int* que anteriormente se había convertido en *bytes* para su envió a través de TCP.

Este proceso, tiene sus versiones correspondientes al resto de variable primitivas dentro de nuestra clase *Packet*. Una vez *packet* se envía o se recibe, *IDisposable* nos permite “reciclar” este objeto para liberar la memoria que este ocupa.

A continuación, se mostrará una representación visual aproximada de este proceso:

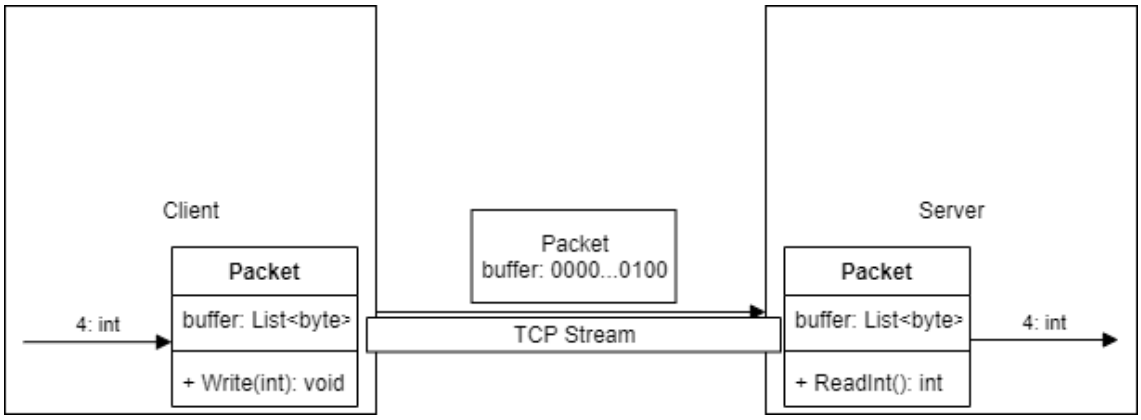


Ilustración 15: Intercambio de paquetes cliente-servidor

6.1.4 Identificación de acción recibida.

Además de obtener información a través de los paquetes necesitamos saber a qué acción se refiere esos datos. Por ejemplo, si el jugador 1 juega la tercera carta de su mano, aparte de mandar el *index* de la mano correspondiente a esa carta, necesitamos informarle al servidor que ese *index* va relacionado con la acción de jugar una carta.

Para ello, crearemos en ambos cliente y servidor dos *enums* con valores que identificarán las diferentes acciones que el cliente puede hacer. Estos *enums* tiene que ser idénticos en ambos sistemas.

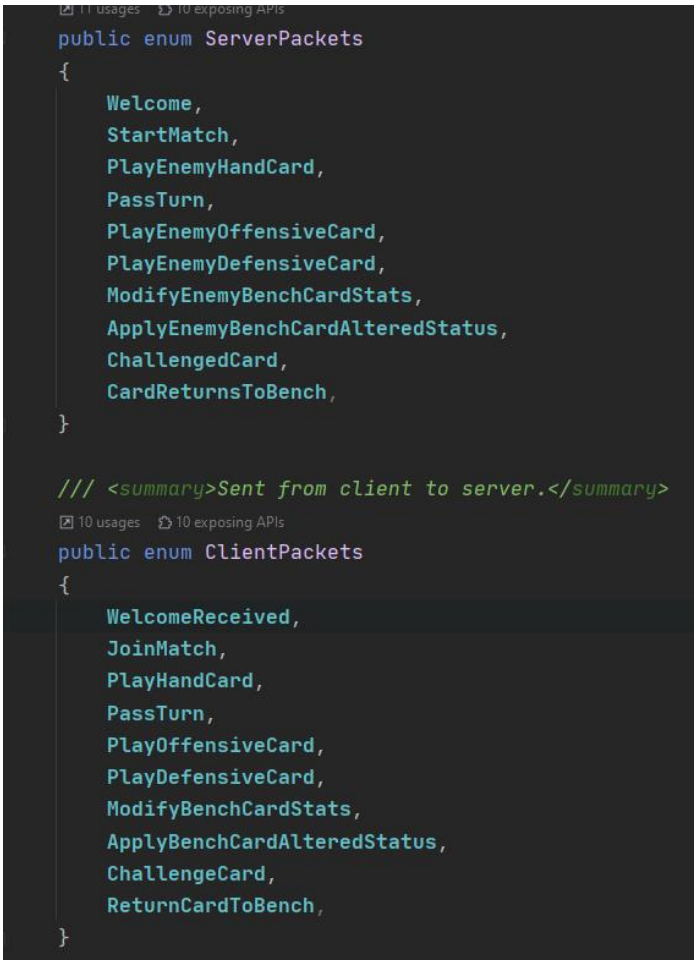


Ilustración 16: Enumerador con los diferentes ids de los paquetes

Siendo *ServerPackets* los *ids* de los paquetes que el servidor envía y *ClientPackets* con los *ids* que los clientes envían.

Estos valores, se castearán como un *int* y se añadirán como 4 *bytes* en el buffer del paquete antes de cualquier dato logrando así especificar sobre que está relacionada la información enviada.

Una vez cada paquete tiene su propio identificador, es necesario implementar una manera para que el servidor diferencie cada uno de ellos y responda en consecuencia a cada uno. Para ello, crearemos una variable *delegate PacketHandler* que referenciará al método correspondiente a cada identificador. Y para la asignación de identificador para cada *packetHandler* crearemos *un Dictionary<int, PacketHandler>* que al pasarle el id del paquete correspondiente, nos referenciará al *PacketHandler* con el método deseado. Estos métodos respuesta serán discutidos más adelante.

La asignación entre *id* y posibles *PacketHandlers* se inicializará en el arranque del servidor.

```
packetHandlers = new Dictionary<int, PacketHandler>()
{
    {(int) ClientPackets.WelcomeReceived, ServerHandle.WelcomeReceived},
    {(int) ClientPackets.JoinMatch, ServerHandle.JoinMatchReceived},
    {(int) ClientPackets.PlayHandCard, ServerHandle.CardPlayed},
    {(int) ClientPackets.PassTurn, ServerHandle.PassTurn},
    {(int) ClientPackets.PlayOffensiveCard, ServerHandle.OffensiveCardPlayed},
    {(int) ClientPackets.PlayDefensiveCard, ServerHandle.DefensiveCardPlayed},
    {(int) ClientPackets.ModifyBenchCardStats, ServerHandle.ModifiedBenchCardStats},
    {(int) ClientPackets.ApplyBenchCardAlteredStatus, ServerHandle.AppliedBenchCardAlteredState},
    {(int) ClientPackets.ChallengeCard, ServerHandle.ChallengedCard},
    {(int) ClientPackets.ReturnCardToBench, ServerHandle.ReturnedCardToBench},
};
```

Ilustración 17: Diccionario con la relación id-Método

ServerHandle siendo la clase que contiene todos los métodos relacionados con la recepción de datos.

Siendo una representación gráfica de lo anterior explicado:

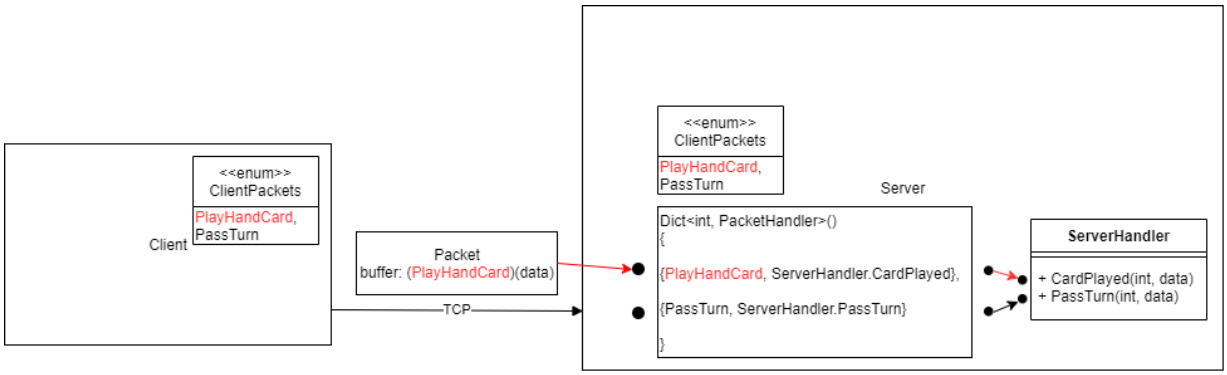


Ilustración 18: Identificación del id del paquete

Los métodos de *ServerHandler* toman como parámetros tanto los datos recibidos como el identificador del cliente emisor para saber quién ha enviado la información y

actuar respectivamente. Teniendo en cuenta que cada *ServerClient* se encarga del enlace de comunicación con su jugador correspondiente, es fácil identificar el emisor.

6.1.5 Envío de paquetes

En el servidor es habitual que los métodos de *ServerHandler* llamen a los de *ServerSend* como respuesta, siendo estos los encargados de crear los paquetes y prepararlos para ser enviados.

Siempre que se envíe un paquete a través de TCP, añadiremos a los 4 primeros *bytes* de cada paquete la longitud de este para que a la hora de la recepción se pueda hacer un recorrido a través de todos los paquetes recibidos y gestionarlos según corresponda.

6.1.6 Procesamiento de información en ServerClient

Para acabar con la implementación de la comunicación del servidor, mostraremos quizás la parte más compleja, la parte relacionada con la manipulación del conjunto de paquetes recibidos y como el servidor lo procesa para utilizar los elementos anteriormente explicados.

Aquí es donde precisamos mostrar el código del método *HandleData(byte[] data)* en *ServerClient*

```
private bool HandleData(byte[] data)
{
    int packetLength = 0;

    receivedData.SetBytes(data);

    //Every packet start starts with an int which indicates the length(4 unread bytes)
    if (receivedData.UnreadLength() >= 4)
    {
        //We store how long the packet lengt through the first int
        packetLength = receivedData.ReadInt();
        if (packetLength <= 0)
        {
            //If the length is <=0 (nothing else to read), we reset the receivedData
            return true;
        }
    }

    while (packetLength > 0 && packetLength <= receivedData.UnreadLength())
    {
        byte[] packetBytes = receivedData.ReadBytes(packetLength);
        ThreadManager.ExecuteOnMainThread( action: () =>
        {
            using (Packet packet = new Packet(packetBytes))
            {
                int packetId = packet.ReadInt();
                Server.packetHandlers[packetId](id, packet);
            }
        });
        packetLength = 0;
        if (receivedData.UnreadLength() >= 4)
        {
            //We store how long the packet lengt through the first int
            packetLength = receivedData.ReadInt();
            if (packetLength <= 0)
            {
                //If the length is <=0 (nothing else to read), we reset the receivedData
                return true;
            }
        }
    }

    if (packetLength <= 1)
    {
        return true;
    }

    return false;
}
```

Ilustración 19: Método para la gestión de datos a través de TCP

Comprobando la longitud de cada paquete podemos saber si en el flujo de información de TCP ya no hay más paquetes que procesar y si los hay, obtener sus

contenidos, entre los cuales, obtendremos el *id* y los datos que a través del diccionario y *delegates* anteriormente explicados son necesarios para que el servidor pueda responder correctamente.

Esta última parte de lectura de *id* y llamada al método correcto lo delegaremos a un hilo en concreto mediante la clase *ThreadManager*. Esto último es necesario para evitar errores inconsistentes.

6.1.7 Implementación de la comunicación en el lado del cliente

La versión de lo anterior explicado en el cliente es muy similar, pero con pequeñas excepciones que cubriremos en esta sección.

- El cliente únicamente tendrá una clase encargada tanto de la conexión con el servidor como el procesamiento de la información que recibe. Esta clase siendo "*Client*".

- Los métodos llamados a la hora de recibir información no precisarán que se les pase el emisor como parámetro ya que siempre será el servidor.

- Los *ids* de los paquetes serán diferentes de los que usa el servidor para identificar la acción. Se creará un *enum* llamado *ServerPackets* con los diferentes *ids* relacionados con los diferentes paquetes que el servidor puede enviar.

El servidor se encargaba de escuchar conexiones mientras que los clientes son los que las establecían. Debido a ello, en la lógica relacionada con la creación del socket que vincula con el servidor, hay que especificar la IP pública y el puerto indicado en al parte del servidor.

6.1.8 Configuración de IPs y puertos.

Para que se establezca la conexión es imprescindible conocer la IP pública del dispositivo en el que se encuentra el servidor, para ello, simplemente accedemos a cualquier página web diseñada para proporcionarla. Como, por ejemplo:

https://www.whatismyip.com/what-is-my-public-ip-address/

Esta ip es dinámica a no ser que se contacte al proveedor de servicios de internet para contratar una ip publica estática que normalmente conlleva un pago extra. Debido a la pequeña magnitud del proyecto se ha decidido mantener la ip dinámica.

Una vez los clientes saben la ip pública, es necesario decidir qué puerto abrir y como abrirlo en la máquina del servidor. En caso de querer publicar este producto se debería revisar la lista de puertos comúnmente usados para evitar usar uno ocupado por otra aplicación. Para ello, existen paginas como:

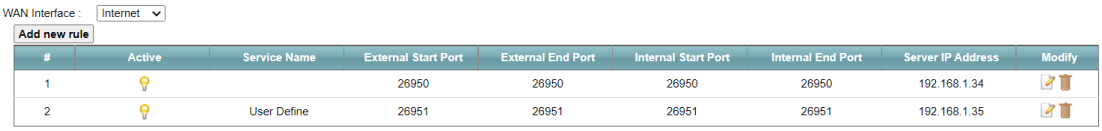
https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

Durante desarrollo y presentación del proyecto se ha usado el puerto 26950 que a continuación enseñaremos como abrir en este caso en concreto.

El proceso de apertura de puertos depende del rúter al cual la máquina del servidor está conectada. En esta situación el rúter es de Movistar modelo HGW-2501GN-R2.

Para acceder a las opciones de configuración, pondremos la ip localhost 192.168.1.1 en cualquier navegador web e introduciremos la contraseña indicada en el rúter.

Una vez dentro, accederemos a la sección *Nat* > *Port Forwarding* y abriremos el puerto 26950 para la ip privada actual en la máquina. Para saber esta ip privada, introduciremos el comando *ipconfig* en la consola de comandos y buscaremos la *IPv4 Address.*


WAN Interface : Internet

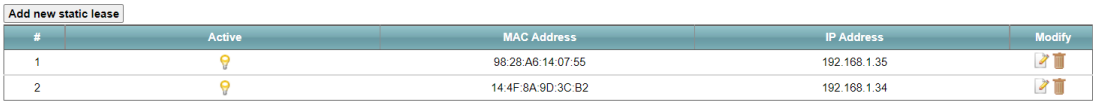
Add new rule

#	Active	Service Name	External Start Port	External End Port	Internal Start Port	Internal End Port	Server IP Address	Modify
1			26950	26950	26950	26950	192.168.1.34	
2		User Define	26951	26951	26951	26951	192.168.1.35	

Ilustración 20: Apertura de puertos.

En este caso, se ha abierto el puerto para ambas ips privadas acabadas en 34 y 35, correspondientes a si la máquina está conectada a través de Wi-fi o ethernet. Para la ip correspondiente con ethernet se ha abierto el puerto 26951.

Debido a que las ips privadas de una red se asignan dinámica mediante el protocolo DHCP[28] tenemos que asegurar que nuestro servidor siempre mantiene las mismas ips privadas las cuales, tienen el puerto asignado abierto. Para ello, en la configuración del router, buscaremos la sección *LAN > Static DHCP* y junto a las direcciones *MAC* de las tarjetas de red de la máquina del servidor asignaremos ips privadas estáticas.


Add new static lease

#	Active	MAC Address	IP Address	Modify
1		98:28:A6:14:07:55	192.168.1.35	
2		14:4F:8A:9D:3C:B2	192.168.1.34	

Ilustración 21: Establecimiento de ips privadas estáticas.

6.1.9 Creación de partidas y emparejamiento de jugadores

Una vez tenemos la información para conectar un cliente con el servidor, este último llevará un recuento de clientes conectados añadiéndolos en dos listas, una para llevar el recuento total y otra para guardar los clientes que están esperando por otro jugador para hacer una partida. Cada vez que un nuevo cliente conecte y se compruebe que hay por lo menos un cliente esperando, el servidor creará un objeto *MatchManager* con el recuento de ambos jugadores y les enviará un mensaje para que las partidas de ambos empiecen. Una vez tenemos agrupados dos clientes en una partida podemos saber que, si un cliente envía información, el destinatario será el otro jugador dentro de la partida.

6.1.10 Conclusiones de etapa

Aunque hayamos abordado en grandes rasgos la implementación de la conexión, volveremos a mencionar elementos de esta en la etapa de implementación del videojuego debido a que nuevas mecánicas conllevan su implementación en esta parte. Por ejemplo, identificar cual es el cliente correcto al que mandarle los datos que se reciben desde su oponente.

---

[28] Protocolo de administración de red encarga de asignación de ips privadas en dispositivos conectados en una red.
Más información: <https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol>

6.2 Segunda Etapa: Desarrollo del videojuego de cartas

Antes de comenzar con la explicación del desarrollo conviene mostrar como el proyecto se ve en su estado final para explicar a continuación como ha sido desarrollado y estructurado.

Ilustración 22: Ilustración general del proyecto

6.2.1 Estudio previo

Antes de comenzar a desarrollar el videojuego se necesita hacer una indagación en como otros videojuegos de este género están estructurados dentro del editor. Una de las primeras preguntas que surgen es si el juego se encuentra en un entorno 2D o 3D.

Buscando información se ha descubierto que, por ejemplo, la escena del tablero de Hearthstone consiste en un entorno 3D visto desde una cámara con proyección ortográfica[29] perpendicular al tablero.

[29] Tipo de cámara que elimina cualquier tipo de perspectiva. Útil para juegos 2D o isométricos.

Ilustración 23: Muestra de Hearthstone en el editor Unity

Teniendo esto en cuenta, se ha recreado en nuestro proyecto la misma situación.

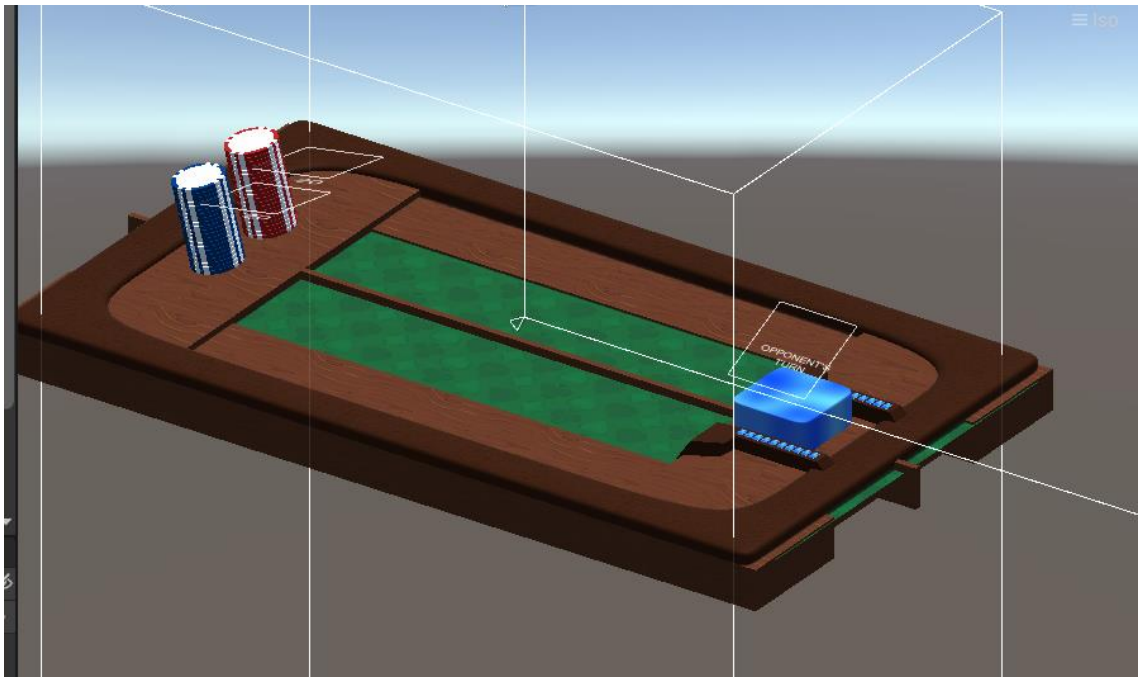


Ilustración 24: Muestra del proyecto en el editor Unity

Aunque los juegos de este género compartan similitudes, existen diferencias en los enfrentamientos. Por ejemplo, las cartas de Hearthstone pueden estar o en la mano o en la mesa, interactuando de diferente manera que, por ejemplo, Runeterra donde existe

una situación intermedia entre la mano y la mesa llamada banco. Estas diferencias conllevan un ritmo diferente en cada juego y es necesario tomar una decisión de que estructura seguirá nuestro proyecto.

Tras pruebas intentando implementar estructuras semejantes a los juegos mencionados como ejemplos se ha decidido usar a Legends of Runeterra como ejemplo de diseño a seguir.

A la hora de crear cartas específicas que hagan cosas en concreto, sea dar un escudo a otra carta, se han escogido efectos de cartas que se pueden encontrar comúnmente en cualquiera de estos juegos.

6.2.2 Estructura del tablero

Como se ha mencionado con anterioridad, el tablero seguirá una estructura similar a la de Runeterra dividiendo cada mitad del tablero en: *Hand*, *Bench* y *Table*. Una representación visual de esta división es:

Ilustración 25: Muestra estructural del tablero

Según el lugar donde se encuentren, las cartas cambiarán visualmente su *layout* e interactuarán de diferente manera.

Cada parte del tablero tiene un numero definido de casillas invisibles que usaremos para encapsular las cartas y que estas interactúen como es debido. Por ejemplo, *PlayerBench* está compuesto de seis casillas *BenchSlot* las cuales contienen los componentes de colisión[30] y la lógica para seleccionar la carta y acciones que se pueden hacer en ese caso. *PlayerBench* tiene las referencias a estas casillas y será esta quien se encargue de ordenarlas dinámicamente en función de si tienen carta o no.

Para evitar repeticiones en el código, se ha usado herencia en los casos como *PlayerTable* y *EnemyTable* que al ser ambas Table, tienen lógica similar. Una muestra de relación entre clases sobre esto sería:

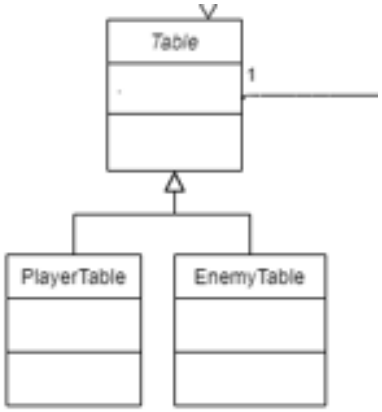


Ilustración 26: Diagrama de clases relacionado con clase Table

[30] Componente invisible encargado de la interacción de físicas. Usado en este caso para detectar el objeto al clicar con el ratón. Más información <https://docs.unity3d.com/ScriptReference/Collider.html>

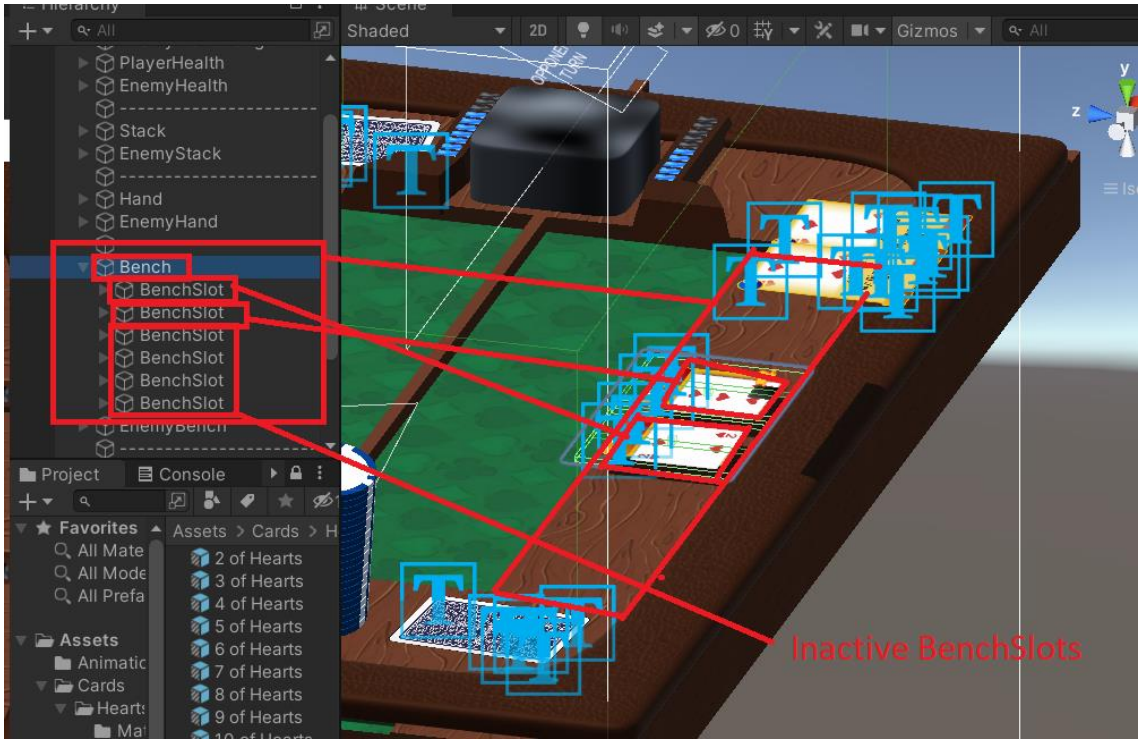


Ilustración 27: Muestra de la división en slots de PlayerBench

En la parte derecha del tablero se puede observar el botón que permite al jugador pasar turno en casos necesarios.

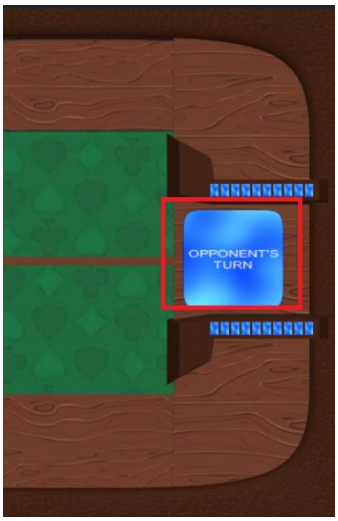


Ilustración 28: Botón para pasar turno

Encima y debajo de este botón están los contadores de energía del oponente y jugador correspondientemente. Este contador está compuesto de diez huecos que se iluminan o no en función de cuanta energía tenga el jugador.

En la primera ronda ambos jugadores tendrán solo uno de energía y cada ronda que pase este valor irá incrementando de uno en uno hasta la ronda diez donde ambos tendrán diez de energía. La energía es usada para poder jugar cartas que estén en Hand.

Por último, a la izquierda se encuentra la vida de ambos jugadores la cual se irá reduciendo según cartas ataquen cuando no hay otra carta defendiendo delate de ellas.

Ilustración 29: Puntos de salud de los jugadores

6.2.3 Fases de una partida

Cada partida está compuesta por un número indefinido de rondas hasta que uno de los jugadores se queda sin puntos, los cuales, están ilustrados a la izquierda del tablero.

Cada ronda está dividida en turnos que se irán intercambiando entre los jugadores en función de las acciones que tomen, por ejemplo, si colocas una carta de *PlayerHand* en *PlayerBench*, pasa a ser turno del oponente para que haga su acción

Cada ronda termina una vez el jugador atacante coloca las cartas atacando y el oponente decide defender o no. Tras eso, la ronda pasará y será el oponente el que podrá colocar cartas atacando.

6.2.4 Posible situación en una ronda

1. Comienza turno, ambos jugadores reciben una carta aleatoria que se coloca en sus respectivas *Hands*

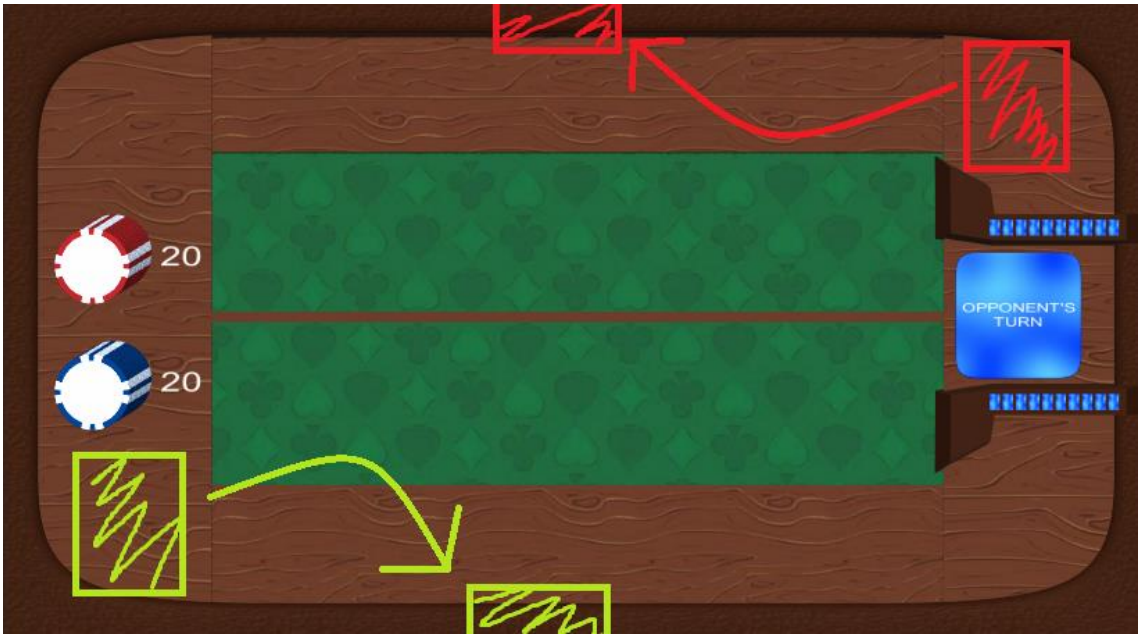


Ilustración 30: Situación ejemplo de ronda 1

2. Si es tu turno, juegas esa carta y esta se coloca en tu *PlayerBench*. Una vez aquí, el oponente puede ver la carta y sabe que puedes atacar o defender con ella. Pasa turno para que el oponente decida si colocar una carta en su *EnemyBench* o pasar turno

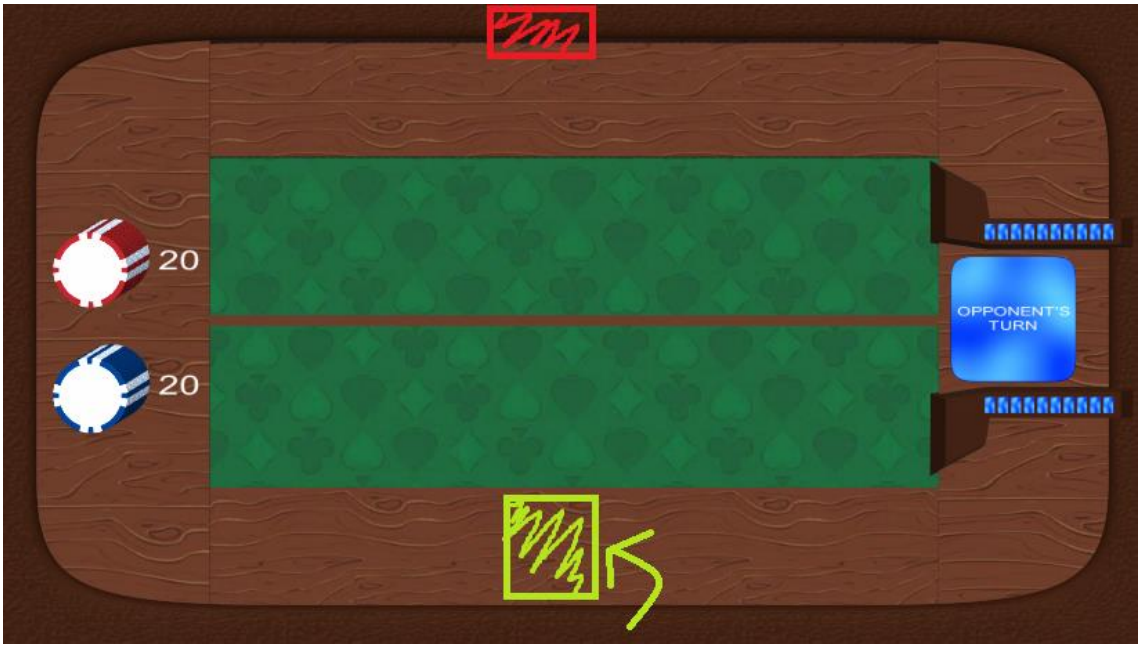


Ilustración 31: Situación ejemplo de ronda 2

3. El oponente decide colocar su carta en *EnemyBench*, pasando así el turno al jugador

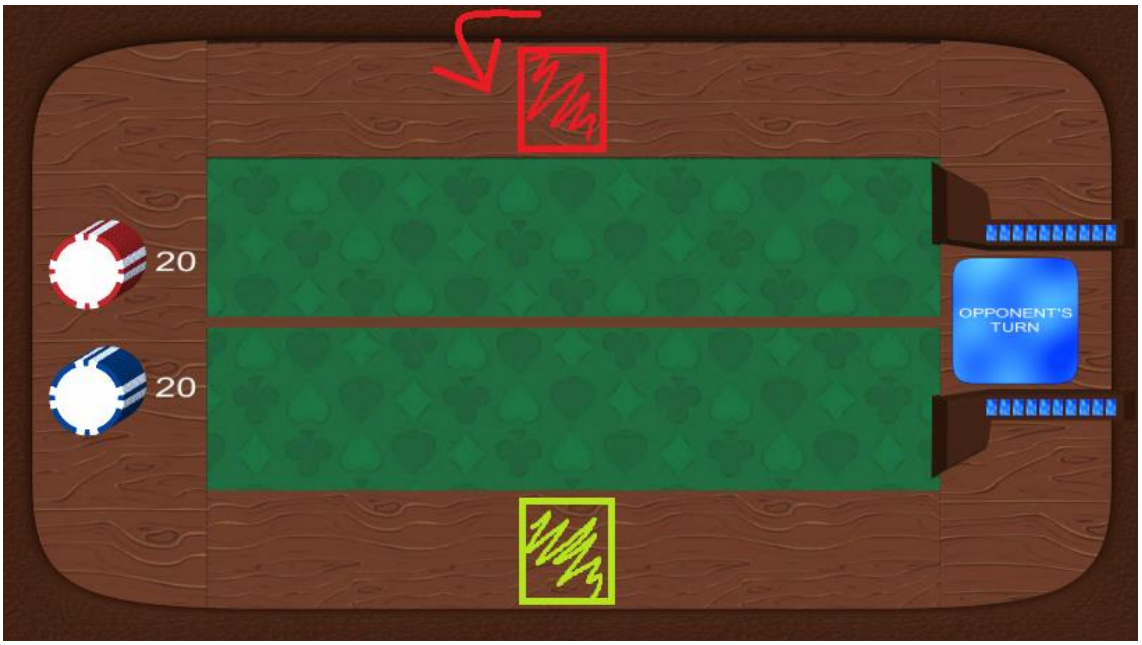


Ilustración 32: Situación ejemplo de ronda 3

4. Si en esta ronda eres tú el atacante, puedes pasar cartas que tengas en el *PlayerBench* al *PlayerTable*. Una vez decidas que cartas van a atacar, pasas turno y el oponente ya no podrá colocar más cartas en *EnemyBench* y tendrá que defender *EnemyTable* con las cartas que ya tiene en *EnemyBench*.

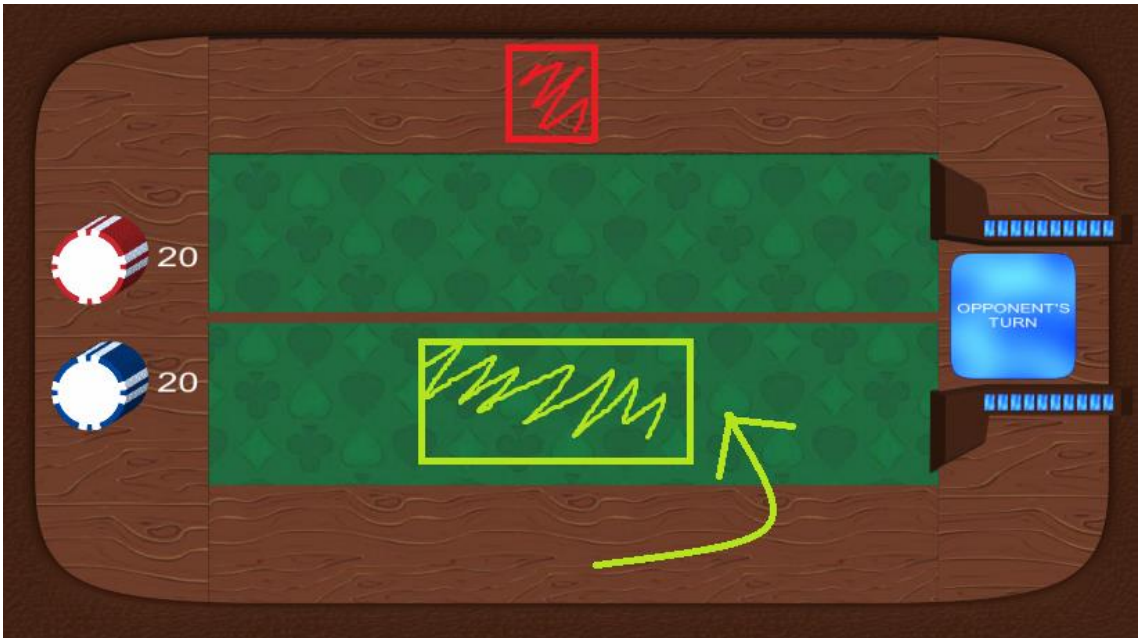


Ilustración 33: Situación ejemplo de ronda 4

5. Si el oponente coloca cartas defendiendo delante de las que atacan, esas cartas chocarán entre ellas y si no, las cartas atacantes golpearán su vida mostrada en la izquierda del tablero. Las cartas que se chocan infligirán su cantidad de daño en la vida de la otra carta opuesta. La ronda pasa, las cartas que hayan sobrevivido vuelven a los *Bench* y es ahora el oponente el que puede colocar cartas atacando.

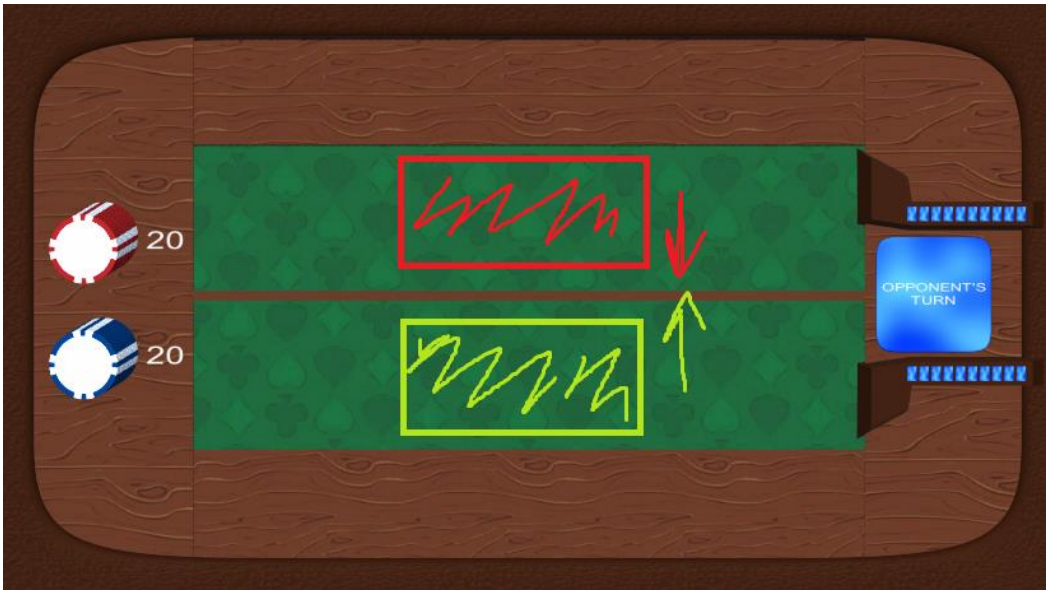


Ilustración 34: Situación ejemplo de ronda 5

6.2.5 Creación del prefab carta

Debido a la existencia de múltiples cartas con diferentes estadísticas y habilidades es necesario crear una plantilla de la cual deriven todas las cartas para evitar crear nuevos objetos de Unity cada vez que se quiera crear una nueva. Para ello, se usan *prefabs*[31]. Con esta herramienta, crearemos un *asset* carta donde definiremos todos los valores generales que lo constituyen y luego crearemos variantes de esta para cada una de las cartas específicas. Por ejemplo, el *prefab* carta "dos de corazones", es una variante del *prefab* general carta y todo cambio que se le haga al *prefab* de carta se aplicará a la variante, pero todo cambio a dos de corazones no se aplicará sobre el *prefab* general.

[31] Sistema de unity que permite crear, configurar y guardar GameObjects completos con todos sus componentes, propiedades y jerarquías dentro de un asset reusable.
Mas información <https://docs.unity3d.com/Manual/Prefabs.html>

Ilustración 35: Prefab carta y sus derivados

De esta manera podremos aplicar cambios a las cartas sin modificarlas una a una.

6.2.6 Estructura de la carta

A continuación, se mostrará la jerarquía del objeto carta.

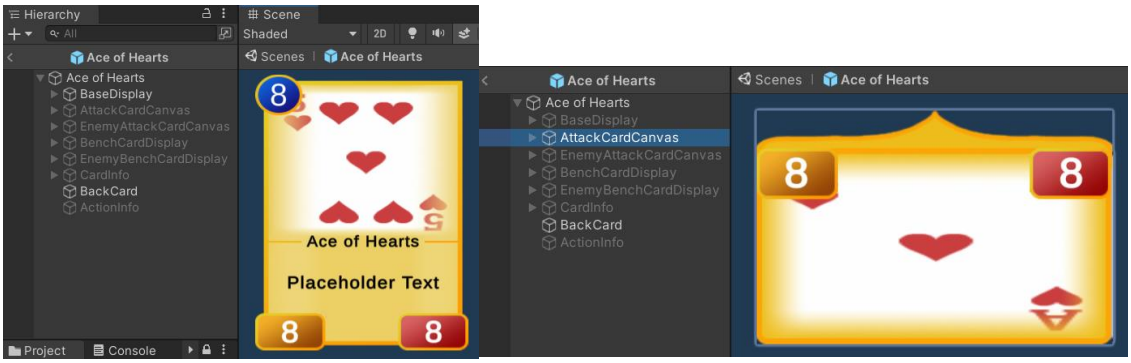


Ilustración 36: Visualización del objeto carta

Como se puede observar, la carta está compuesta de diferentes elementos activados o desactivados que corresponden a como se visualiza en la escena. Dependiendo de donde esté la carta, en la mesa, mano… necesitamos cambiar la información mostrada al jugador para dar retroalimentación de la situación en la que se encuentra la carta.

Cada uno de estos componentes lleva un registro de referencias de todo lo mostrado en ellos:

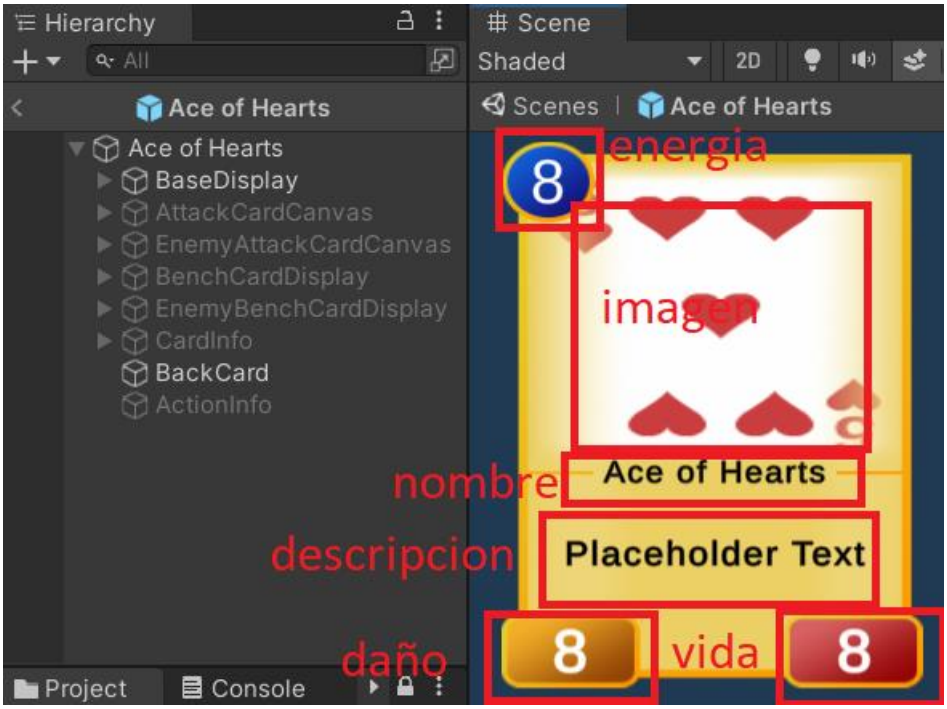


Ilustración 37: Elementos del objeto carta

En algunos casos es necesario mostrar información que en otros no, por ejemplo, cuando una carta es jugada ya no es necesario mostrar cuanta energía cuesta jugarla, por lo tanto, solo se mostrará este dato en la versión de la carta en la mano.

Toda esta información es modificada a través de scripts dentro del objeto y referenciados desde los diferentes elementos de la partida.

Todas las cartas tienen 3 grandes estadísticas en común, siendo estas:

- Energía/Magia: Hace referencia a la cantidad de energía que es necesaria para colocar una carta desde la *Hand* al *Bench*.

- Daño: relacionada a la cantidad numérica que infringirá la carta sobre otras cartas enemigas o vida del jugador al enfrentarse.

- Vida: cantidad de puntos que puede perder la carta antes de que desaparezca del tablero.

Para terminar, hay que aclarar que la representación de las cartas está compuesta de planos 3D con texturas aplicadas que nos permiten simulación de sombras y mayor sencillez a la hora de mover cartas entre ellas y aplicarles efectos.

6.2.7 Interacción del usuario

La mayor parte de la interacción con el videojuego es, si no la única, seleccionando cartas y arrastrándolas estratégicamente al lugar deseado.

A continuación, se explicarán cómo funcionan los componentes que permiten al jugador participar en las acciones.

Teniendo en cuenta los anteriormente mencionados *slots* dentro de las *Hands*, *Benches* y *Tables*, se da uso a sus colisiones y referencias a sus posibles cartas para seleccionar la carta que queremos. Para que el juego sepa que *slot* estamos seleccionando usaremos un componente dentro de las físicas de Unity llamado *Raycast*[32] el cual hará una línea vertical invisible atravesando el escenario que, indicándole un filtro de *layers*[33] y modificando su origen como la posición dinámica del ratón, nos avisará si está en contacto con ciertos *colliders*[34]. Una creación de lo anteriormente mencionado sería:

Physics.Raycast(CameraManager.Instance.mousePosition, out RaycastHit hit, Mathf.Infinity, 1 << 11);

El cual nos devolverá true o false en función si colisiona con *colliders* definidos en la *layer* 11, en este caso, *HandSlots*. Además, nos indicará el elemento colisionado mediante el parámetro *out hit*. Esta información será usada por los slots correspondientes para que cuando se importe un input de ratón, compruebe si es el *slot* seleccionado y mueva la carta según corresponda.

Un ejemplo visual de esta funcionalidad sería:

---

[32] Línea invisible en la escena de Unity para detección de colisiones que requiere un punto origen, dirección, distancia y opcionalmente un filtro de capas.
Mas información <https://docs.unity3d.com/ScriptReference/Physics.Raycast.html>
[33] Medio de Unity para definir que objetos en la escena de Unity pueden interactuar con otras de diferente manera. Mas información <https://docs.unity3d.com/Manual/Layers.html>
[34] Componente de Unity encargado de definir la forma de un objeto con el objetivo de calcular colisiones de físicas. Mas información <https://docs.unity3d.com/Manual/CollidersOverview.html>

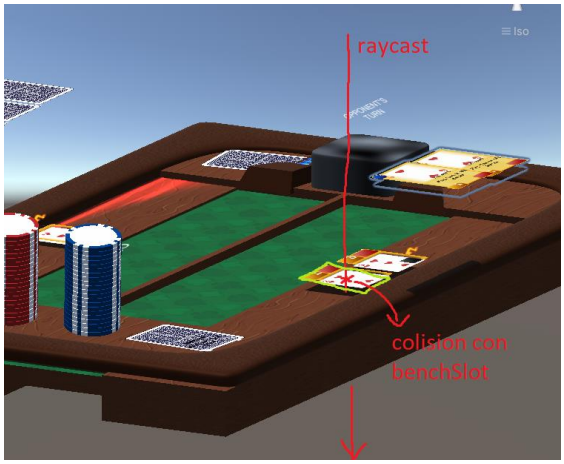


Ilustración 38: Muestra del Raycast y su posible colisión

Tras lo anterior, pongamos como ejemplo que es una carta en un *BenchSlot* la que es seleccionada. Ahora la posición de la carta seguirá el ratón esperando la decisión del jugador que puede ser, por ejemplo, arrastrarla al *collider* de Table y clicar otra vez, así colocándola en un *TableSlot* o podría cancelar la acción haciendo clic derecho devolviendo la carta a su anterior *BenchSlot*. Cartas con habilidades como darle escudo a otra tendrán fases intermedias de *inputs* para seleccionar a que carta darle escudo.

Hay elementos que se deben tener en cuenta para ciertas acciones. No poder jugar una carta de *Hand* la cual cuesta más energía que la que tienes en esa ronda o no poder seleccionar cartas durante el turno del oponente.

Cuando un jugador hace una de estas acciones y mueve una carta de sitio, se tiene que informar al otro jugador y aquí es donde añadiremos un nuevo método en *ClientSend* con información del *slot* donde se encontraba antes la carta y a donde ha ido. Estos datos se empaquetarán con un id nuevo creado, por ejemplo, "*PlayHandCard*" y será enviado mediante TCP al servidor el cual comprobará cual es el oponente e informará a este para que cambie la escena de su cliente y la sincronice con la del emisor.

6.2.8 Reordenación de cartas

Uno de los componentes al cual se le ha dado especial atención es al movimiento de cartas en el tablero y la reordenación dinámica de estas en cada contenedor. Por

ejemplo, si en *Bench* hay una carta y se le añade otra, estas dos cartas se ajustarán dinámicamente para ocupar espacio simétricamente.

Una visualización de lo anteriormente mencionado sería:

Ilustración 39: Una carta en Bench y tres en Hand

Ilustración 40: Dos cartas en Bench y dos en Hand

Para lograr esto, cada contenedor (*Hand*, *Bench*, *Table*) lleva un seguimiento del número de cartas que tienen. Cada vez que se notifica que una carta se ha desplazado a otro contenedor este se encarga de cambiar su posición, escala o rotación según convenga para cada situación.

La principal pieza para conseguir este efecto de manera dinámica es utilizando el *Vector3.Lerp*[35] incluido en las librerías de UnityEngine. Usándolo junto a una iteración dentro de un *IEnumertor* podemos lograr animaciones que mediante el animador de Unity no podríamos tan fácilmente.

Con esto dicho, animaciones como mostrar la mano cuando se pasa el ratón por encima son logradas con el animador debido a que siempre van a ser iguales y no dinámicas.

6.2.9 Paso de turnos y estado de la partida

[35] Método de la librería UnityEngine que interpola linealmente un elemento entre dos puntos. Mas información <https://docs.unity3d.com/ScriptReference/Vector3.Lerp.html>

El *GameManager* es un script dentro de la solución cliente encargado de llevar un recuento de la situación de la partida. Una de sus funciones principales es calcular en qué estado se va a encontrar la partida una vez el jugador haga una acción u otra. Si el jugador 1 juega una carta de *Hand* a *Bench*, el turno se pasa automáticamente al jugador 2. Esto no ocurre igual en el caso de que se jueguen cartas de *Bench* a *Table* ya que el jugador puede decidir añadir más cartas para atacar. Todas estas comprobaciones y resultado son calculados en métodos dentro de *GameManager*, *PassTurn* en el caso de que el jugador haga acción o *TurnPassed* para actualizar el estado de la partida en función de cómo ha actuado el oponente.

Sin entrar en detalle de todo lo que se comprueba, este sería el código correspondiente a *PassTurn*

```
public void PassTurn()
{
    if (currentState == GameState.PlayerTurn)
    {
        onPassedTurn?.Invoke();
        EnemyTable.Instance.ResetSlotsChallenged();
        ChangeGameState(GameState.EnemyTurn);
        CheckTurnStatus();
        CheckAllCardsMana();

        if (PlayerTable.Instance.CheckCardsOnTable())
        {
            hasAttackToken = false;
            ChangeGameState(GameState.EnemyDefendTurn);
            PlayerTable.Instance.CheckCardsUsability();
            ClientSend.PassTurn();
            if (hasBenchedCard)
                hasBenchedCard = false;
            return;
        }

        if (hasBenchedCard)
        {
            ClientSend.PassTurn();
            hasBenchedCard = false;
            return;
        }

        if (hasSkippedAttack)
        {
            ButtonManager.Instance.ChangeButtonStatus( value: false,  text: "Opponent's Turn");
            playerBenchEffect.SetBool( name: "Turn",  value: true);
            enemyBenchEffect.SetBool( name: "Turn",  value: false);
            hasSkippedAttack = false;
            hasAttackToken = true;
            ClientSend.PassTurn();
            StartCoroutine( routine: RoundReset(GameState.PlayerTurn));
            return;
        }

        if (hasAttackToken)
            hasSkippedAttack = true;
        ClientSend.PassTurn();
        return;
    }
```

Ilustración 41: Método encargado del paso de turno

La gran cantidad de comprobaciones que tienen lugar durante el paso de turno conllevan a varios posibles escenarios que se mostrarán a través de un *FlowChart* a continuación.

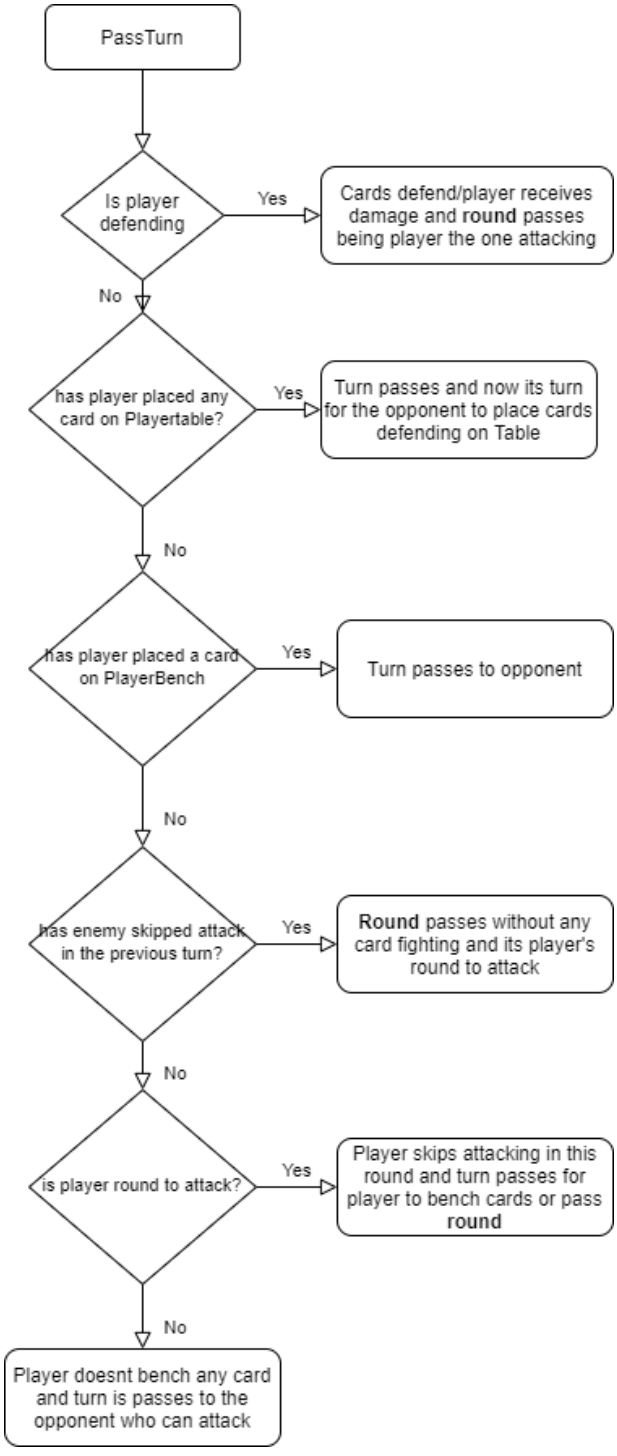


Ilustración 42: FlowChart de las condiciones del paso de turno.

Este recorrido tiene que ser traducido siguiendo las acciones del oponente para asegurarnos de que ambos clientes estén sincronizados.

6.2.10 Cartas y habilidades especiales

En este género de videojuegos necesitamos que cada carta sea única más allá de sus estadísticas de daño, vida y energía. Es necesario dotar a cada carta de alguna habilidad especial que la diferencie del resto.

Para este proyecto se han hecho varios ejemplos de cartas con esto en mente, las cuales, se nombrarán y se explicarán brevemente a continuación:

- **Two of Hearts | *Play: Give an ally Barrier***

Cuando jugada desde Hand a *Bench*, la carta pedirá escoger una carta aliada que esté situada en el *Bench*. Una vez seleccionada, la carta seleccionada obtendrá un efecto que prevendrá que reciba daño la próxima vez que golpee y *Two of Hearts* se colocará en el *Bench*. En caso de que no haya ninguna carta anteriormente en *Bench*, *Two of Hearts* se colocará directamente en *Bench* perdiendo la posibilidad de usar su habilidad.

- **Four of Hearts | *Play: Grant an ally +2 | +2***

Siguiendo un esquema similar a *Two of Hearts*, esta carta selecciona una carta en el momento de ser jugada en *Bench* añadiendo a la carta seleccionada un incremento de daño y vida en sus estadísticas de 2. Por ejemplo, una carta con 1 de daño y vida, tendría 3 de daño y vida una vez seleccionada.

- **Five of Hearts | *Strike: Double my damage***

Esta carta, activa su efecto en el momento que se encuentra en *Table* y golpea a otra carta o al jugador. Tras esa acción, la estadística de daño de la carta se duplica.

Estas son las cartas que nos sirven como ejemplo en este proyecto de como el sistema de cartas con habilidades únicas funcionaría.

Además de estas habilidades únicas, existen otros efectos que se pueden encontrar en diferentes cartas. Por ejemplo:

- **Challenger**

Permite a la carta que lo tiene forzar a una carta enemiga a defender su ataque sin que el contrincante pueda hacer nada al respecto.

Una vez jugada la carta con *Challenger* en *PlayerTable*, se le permitirá al jugador seleccionar y arrastrar la carta enemiga del *EnemyBench* y esta se colocará defendiendo la carta con este efecto.

Este efecto es indicado en la carta mediante los siguientes iconos marcados con un recuadro verde

Ilustración 43: Iconos del efecto Challenger

Para facilitar la programación de estas cartas se ha usado herencia para que no sea necesario implementar el mismo código para cartas que tienen lógica similar, por ejemplo, cartas que tienen que actuar cuando son jugadas en *PlayerBench.*

El diagrama de clases relacionado con esta relación es el siguiente:

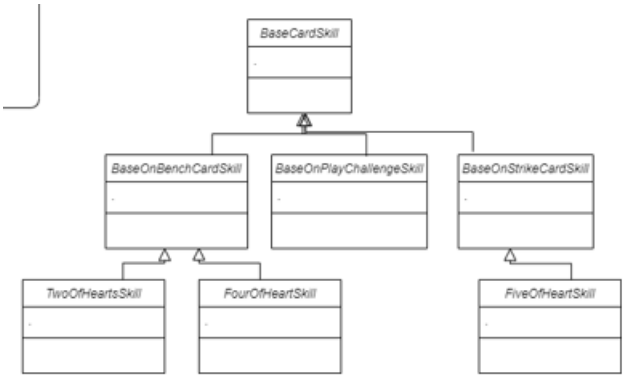


Ilustración 44: Diagrama de clases de la lógica de la carta

6.2.11 Elementos para la retroalimentación[36]

En un videojuego es necesario darle información al jugador para que este sepa que está pasando y que puede hacer. Por ello, se ha dado especial atención a pequeños detalles para hacer la jugabilidad de este proyecto más intuitiva.

Muchos de estos elementos están creados mediante la herramienta de *ShaderGraph* incluida en Unity.

A continuación, se hará un listado de los diferentes elementos, principalmente visuales, implementados en esta solución:

- **Efectos de turnos en el tablero:**

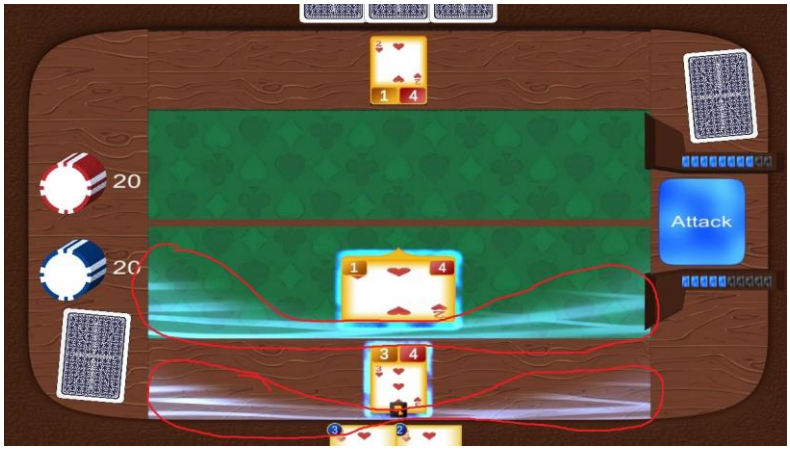


Ilustración 45: Muestra del efecto de turno del tablero

[36] Reacciones de un videojuego como consecuencia de las acciones de un jugador.

Este efecto es activado para, en el caso del *Bench*, indicar si el turno es del jugador u oponente y, en el caso de Table, indicar que una carta ha sido jugada en esta.

Este efecto ha sido creado a través de *ShaderGraph* y su *shader*[37] se compone, sin entrar en detalles, tal que:

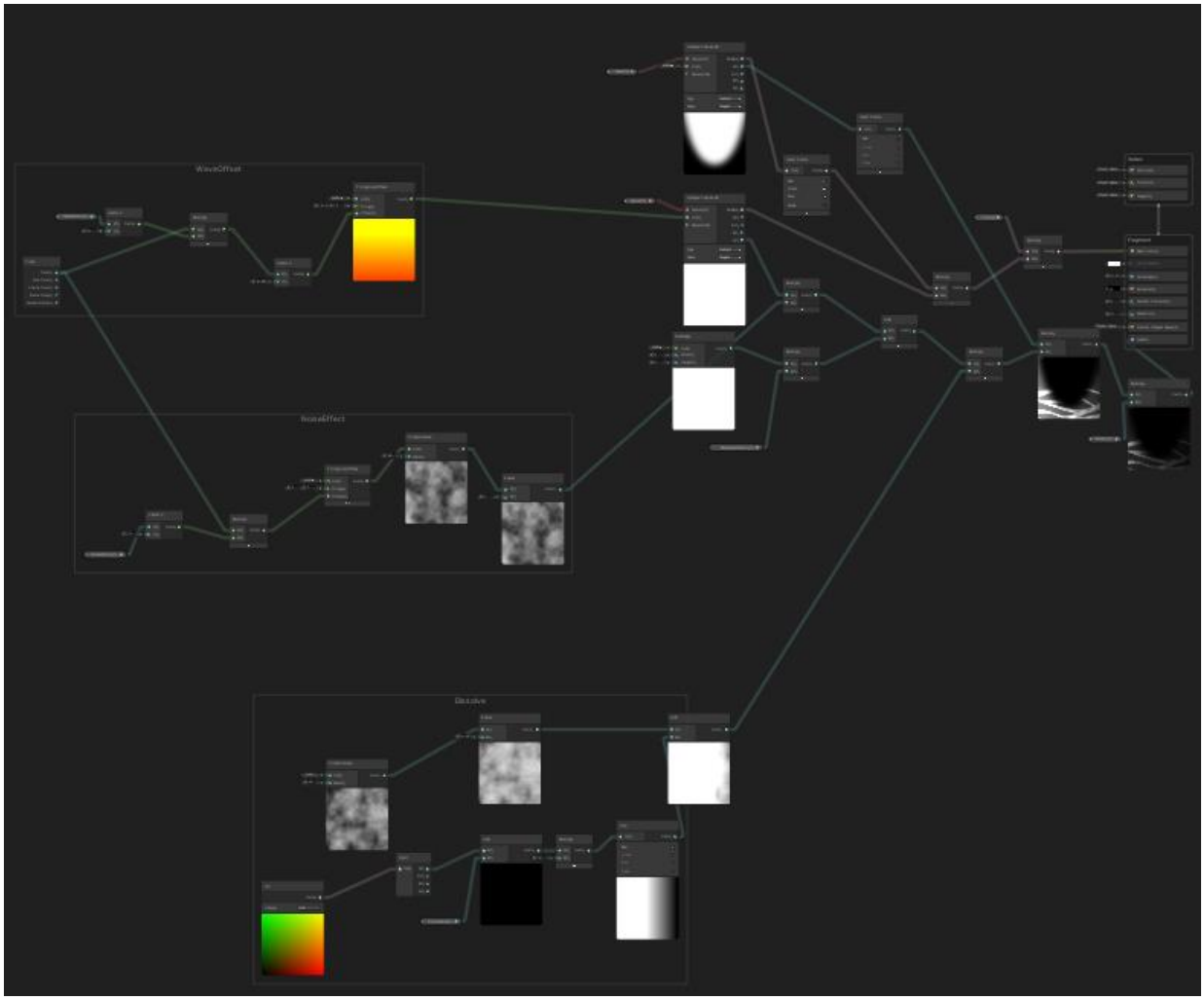


Ilustración 46: Shader en ShaderGraph del efecto turno del tablero

Debido a que los *shaders* se han considerado un apartado más gráfico en el proyecto, se ha decidido no profundizar en la función de cada nodo en los *shaders* y se han mostrado las ilustraciones de estos únicamente para mostrar la extensión y cantidad de nodos que los componen.

[37] Clases que especifican las variables y cálculos que una textura tiene que usar dentro de un motor gráfico.

- **Efectos de usabilidad de las cartas:**

De la misma manera que se ha creado el efecto del tablero, se ha creado un efecto que se activará en función de si se puede interactuar con una carta o no.

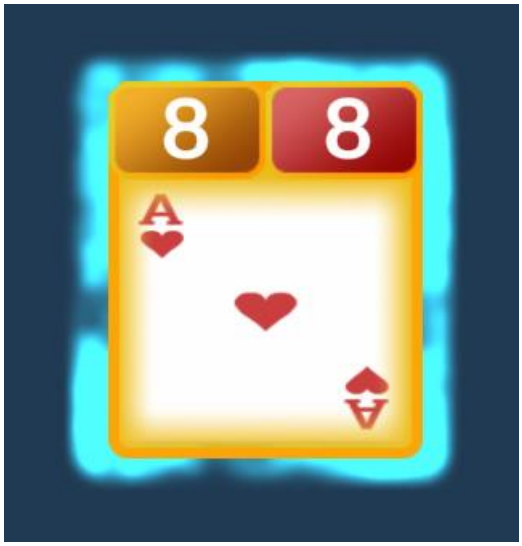


Ilustración 47: Efecto de carta usable

De esta manera, el jugador siempre sabe qué y que no puede clicar.

Siendo su *shader*:

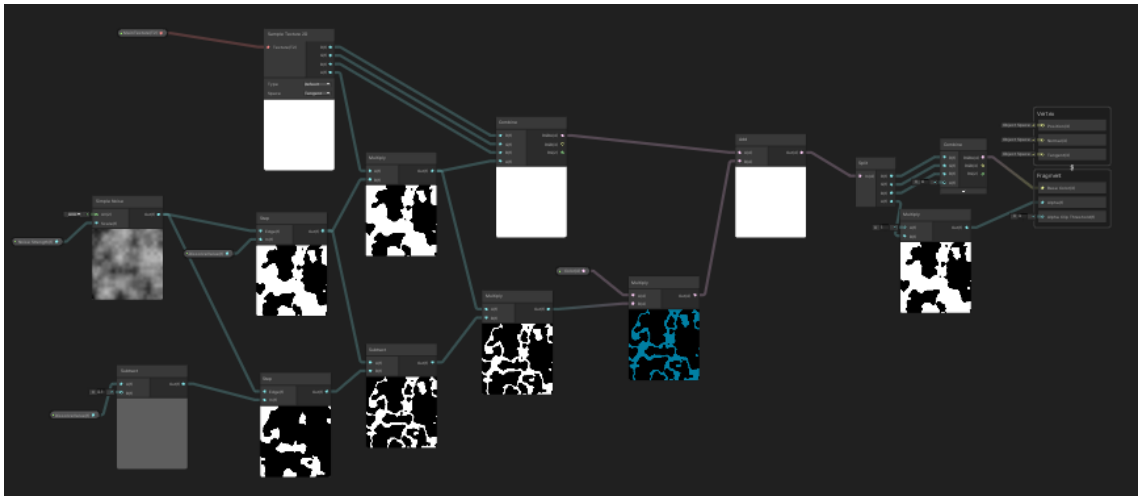


Ilustración 48: Shader en ShaderGraph del efecto de carta usable

- **Animaciones:**

El desplazamiento de las cartas por el tablero está animado con diferentes movimientos para facilitar al jugador a donde van las cartas jugadas.

Estas animaciones son creadas tanto a través de código para las situaciones dinámicas como usando el *Animator* dentro de Unity para casos estáticos. Como ejemplo de animación estática es cuando las cartas golpean a otras cartas o jugador.

- **Botón e indicadores de energía**

El botón para pasar turno también está compuesto de un efecto creado a través de *ShaderGraph* para saber cuándo se puede pulsar además de tener un texto que cambiará informando del estado de la partida, como, por ejemplo, indicando que es "*Opponent's Turn*".

Ilustración 49: Botón activado y usable

Ilustración 50: Botón desactivado

Además, los indicadores de energía encima y debajo del botón informan a los jugadores de cuanta energía les queda en esa ronda.

Ilustración 51: Jugador con 8 de energía, oponente con 10 de energía

- **Mensajes emergentes**

Junto a la información que proporciona el botón, mensajes en medio del tablero informarán a los jugadores de la situación de la partida. Como ejemplos, "*Round 2*", "*Your Action*".

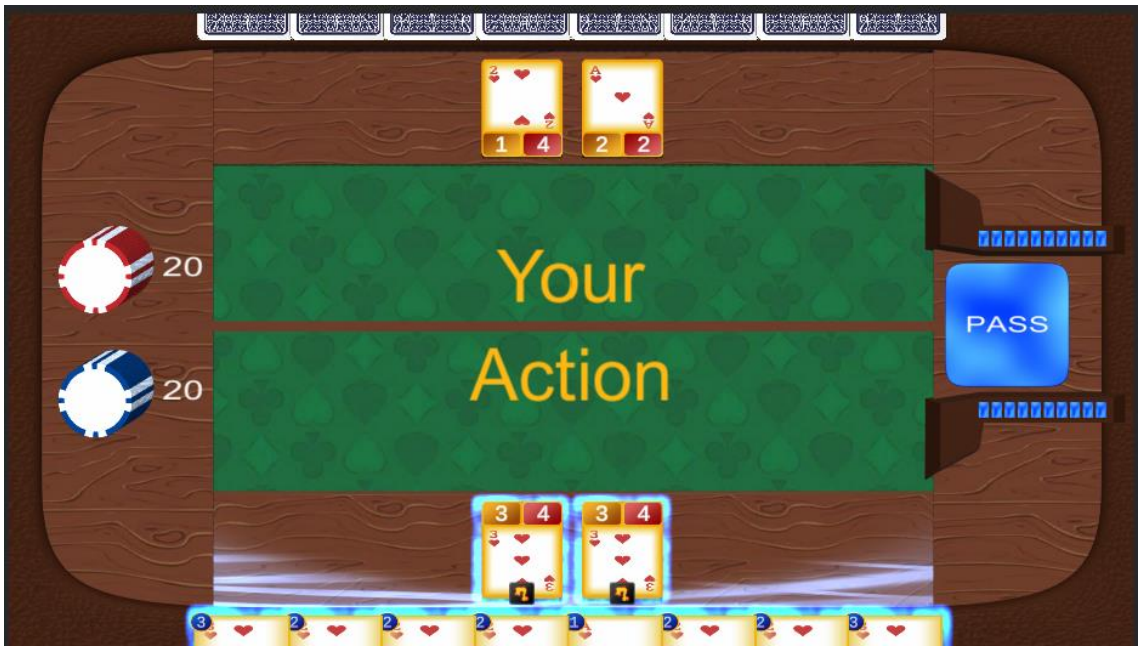


Ilustración 52: Ejemplo mensajes emergentes

- **Descripción emergente de cartas**

Si el usuario mantiene el cursor estático durante un segundo encima de alguna carta que esté en un *Bench* o *Table*, aparecerá una imagen a su derecha con toda la información posible respecto a esa carta.

Ilustración 53: Ejemplo de descripción de carta

- **Efectos de habilidades que necesiten escoger otra carta**

Algunas cartas precisan de la selección de alguna otra carta para llevar a cabo su habilidad. Para facilitar está acción al jugador se han añadido elementos como texto explicativo y líneas animadas con efectos que hacen de enlace visual entre la carta jugada y la escogida.

Ilustración 54: Ejemplo de efecto de selección de carta aliada

Dentro de esta sección hay que hacer especial mención a los efectos visuales relacionados con el efecto de *Challenger*.

Ilustración 55: Ejemplo de efecto Challenger

Como se puede observar en la ilustración anterior, la carta en *PlayerTable* con el efecto *Challenger* tiene un brillo naranja a diferencia de la carta sin este efecto. Además, las cartas en *EnemyBench* tienen el mismo efecto junto a una ilustración indicando que

el jugador tiene la posibilidad de seleccionar cualquiera de esas cartas y arrastrarla para que defienda enfrentándose a la carta con *Challenger* de *PlayerTable.*

Una vez seleccionemos y arrastremos la carta enemiga que queramos, una animación aparecerá confirmando la acción de la habilidad *Challenger.*

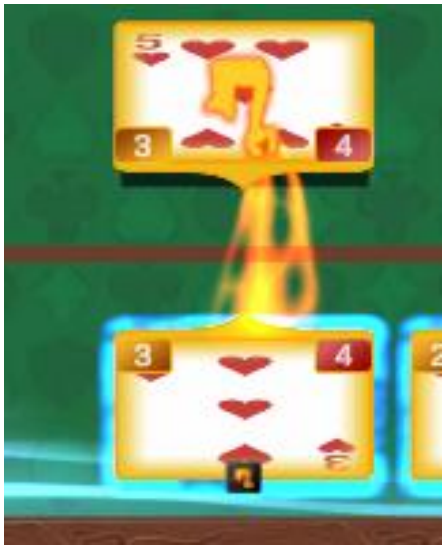


Ilustración 56: Animación de la habilidad Challenger

Esta animación será presente también en el cliente del contrincante para informarle de la acción *Challenger* tomada por el jugador opuesto a él.

- **Efecto anterior a colocar una carta**

Para ayudar al usuario saber a dónde va a ir la carta seleccionada, se ha añadido un efecto que brillará en *PlayerBench* o *TableBench***.** Además, este efecto aparecerá al mismo tiempo en el cual se reordenan las cartas de los dos lugares anteriormente mencionados para mostrar cómo sería la situación del escenario tras hacer la acción de colocar la carta.

Siendo para el caso de *PlayerBench* antes:

Ilustración 57: Mesa antes de la muestra de efecto anterior a colocar carta

Y después:

Ilustración 58: Muestra de efecto anterior a colocar carta

Un efecto similar se puede ver en *PlayerTable:*

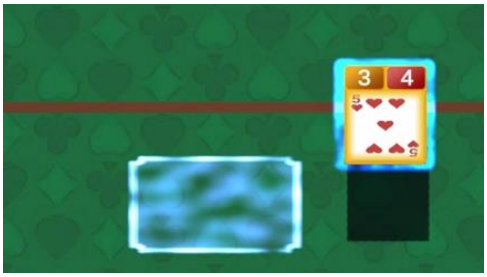


Ilustración 59: Muestra de efecto anterior a colocar carta en PlayerTable

7 Pruebas

Durante el desarrollo del proyecto se han ido haciendo pruebas para comprobar que las mecánicas implementadas funcionen correctamente y no choquen con otros aspectos de la solución.

La manera más frecuente para testear ha sido simular partidas de dos jugadores jugadas en el mismo ordenador por la misma persona. De esta manera, se han podido comprobar componentes relacionados con el videojuego como, por ejemplo, movimiento de cartas.

Gracias a estas pruebas se han podido hacer correcciones a medida que se han desarrollado las implementaciones.

Para asegurar que el sistema multijugador funciona entre diferentes dispositivos, se ha enviado el lado del cliente a otros ordenadores fuera de la LAN donde se encuentra el servidor. Haciendo pruebas de esta manera se ha buscado la manera de configurar las IPs y puertos de manera que otros usuarios puedan conectarse con el servidor.

Se han llevado a cabo pruebas con más de dos personas para ver la capacidad del servidor de crear partidas sin errores a la hora de conectarse.

8 Conclusiones

Durante la realización de este proyecto se han encontrado diferentes obstáculos a los cuales se les ha tenido que encontrar alternativas. Entre ellos, la búsqueda de una estructura que de base a lo que sería la comunicación online, decidirse por una estética específica para simplificar la búsqueda de ilustraciones para las cartas, que videojuego se tomaría como referencia para implementar el sistema de manera más clara para la exposición futura y delimitar la cantidad de contenido a implementar que fuese suficiente para que el videojuego se sintiese suficientemente completo para considerarse acabado para esta tarea.

A las cuestiones anteriores se les ha ido proporcionando soluciones durante el desarrollo como, por ejemplo, el uso de ilustraciones de cartas regulares para evitar cualquier problema relacionado con derechos de autor, un estudio exhaustivo más allá de lo aprendido en la carrera relacionado con comunicación TCP, librerías que faciliten su implementación, Unity3D y C#.

Tomar como referencia un videojuego el cual el desarrollador está más familiarizado con ha sido de gran ayuda a la hora de tomar decisiones de diseño y ha ahorrado mucho tiempo que ha sido invertido en la implementación de características que han solidificado al producto final.

Uno de los factores que más ha condicionado el desarrollo ha sido tomar decisiones sobre que es necesario y que no para considerar el proyecto finalizado. Decisiones como implementación de efectos que ayuden al jugador a entender lo que está pasando en la partida y cuando es su turno se han considerado indispensables, en cambio, factores como sonidos y música que únicamente mejoran la experiencia del usuario y no son necesarias para el entendimiento del proyecto, se han relegado a futuro desarrollo. Además, el número inicial de cartas pensado antes de la realización del proyecto era mucho mayor, pero a medida que se ha ido desarrollando y considerado la magnitud de contenido a implementar, se ha decidido hacer un número reducido de cartas suficientes para la exposición de este proyecto.

Estas cuestiones han sido factores que han condicionado el resultado final, pero sin afectar ninguno de los objetivos iniciales planteados al principio del desarrollo.

8.1 Relación del trabajo desarrollado con los estudios cursados

La realización de este proyecto ha requerido el uso de múltiples tecnologías que han sido enseñadas a lo largo de la carrera. A continuación, haremos un recorrido de que se ha usado y su relación con las asignaturas cursadas durante la carrera:

- **Ingeniería de Software | Programación | Estructuras de datos y algoritmos**

Asignaturas indispensables que han sido punto de partida para el conocimiento de conceptos relacionados con programación como paradigmas como programación orientada a objetos (POO) con Java y C# utilizados durante el proyecto. Se han usado cosas aprendidas desde lo más básico aprendido referente a tipos de variables hasta polimorfismo, gestión de objetos y patrones de diseño.

- **Sistema y servicios en Red | Redes de computadores**

Asignaturas que ha enseñado conceptos de la pieza principal de este trabajo. Conceptos relacionados con la transmisión de datos y protocolos como TCP y UDP. Una introducción a estas tecnologías que ha supuesto un punto de partida para la creación de una aplicación útil.

- **Introducción a la programación de videojuegos | Entornos de desarrollo de videojuegos | Animation and design of videogames.**

Segunda pieza fundamental de proyecto. Estas asignaturas han enseñado que es posible lograr con la herramienta que es Unity3D y cómo lograr productos desde videojuegos hasta aplicaciones menos centradas al ocio. Uso de los diferentes elementos de Unity para videojuegos 2D y 3D, buenas prácticas y conceptos relacionados con animaciones y patrones para crear resultados de calidad.

Además, un punto de partida para enfocar la programación a un entorno más interactivo y saber las diferentes pautas a seguir para la creación de un videojuego completo.

- **Diseño y configuración de redes de área local**

Necesaria para la configuración y manipulación de IPs y puertos para lograr la conexión cliente-servidor.

- **Diseño y Modelado 3D**

Utilizada en menor medida, pero uso para la creación de apartados visuales dentro del apartado del videojuego.

Estas son las principales asignaturas que han tenido mayor repercusión durante el desarrollo, pero si se llegase a continuar el desarrollo, deberemos tener en cuenta otras asignaturas que nos han proporcionado conocimientos útiles para un producto más elaborado, sean estas:

- **Asignaturas relacionadas con organización de empresas**
- **Bases de datos y sistemas de información**
- **Tecnología de bases de datos**
- **Estadística**

9 Trabajos futuros

La manera más común de monetización en este género de videojuego es mediante paquetes de nuevas cartas para que los jugadores monten sus propias y nuevas barajas. Por lo tanto, una de las primeras necesidades como desarrollador es pensar en la implementación de nuevos tipos de cartas para expandir las posibilidades para los jugadores.

Además, existen otros elementos que es necesario planear para un futuro próxima si se quisiese seguir adelante con el desarrollo. Listadas a continuación:

- **Base de datos con información de los jugadores**

Teniendo en cuenta la faceta competitiva del videojuego es necesario pensar en la creación de una base de datos donde se actualicen los perfiles de los jugadores donde guardaremos información como que cartas tienen desbloqueadas, mazos que hayan creado, un historial de partidas y algún sistema de puntuación y progreso para emparejar a los jugadores con otros con una habilidad semejante.

- **Sonido y música**

Una de las piezas fundamentales de un videojuego es su apartado sonoro. Ya que nos hemos centrado en la implementación del apartado multijugador y la jugabilidad, se ha decidido proponer esta parte del producto como algo a implementar si se quisiera seguir desarrollando de manera comercial para futuros usuarios reales.

- **Expandir el equipo**

Este proyecto ha sido realizado por una persona, así que, para conseguir los planes anteriores y pensando en un desarrollo a largo plazo lo óptimo sería buscar gente especializada en arte, audio, diseño y marketing para conseguir un resultado de mayor calidad y aumentar la velocidad de producción.

10 Referencias bibliográficas

[Will Roya] The History of Playing Cards: The Evolution of the Modern Deck, de https://playingcarddecks.com/blogs/all-in/history-playing-cards-modern-deck 16 de octubre de 2018

[Sonam Adinolf y Selen Turkay] Collection, creation and community: a discussion on collective card games, de https://www.researchgate.net/publication/262326293_Collection_creation_and_community_a_discussion_on_collectible_card_games junio de 2011

[Glenn Fiedler] Networking for Game Programmers UDP vs TCP, de https://gafferongames.com/post/udp_vs_tcp/ 1 de octubre de 2008

[Alan Thorn] Unity 4 fundamentals: get started at making games with Unity, New York: Focal Press 1st Edition, 2014

[Daniel Pittman y Chris GauthierDickey] Match+Guardian: A Secure Peer-to-Peer Trading Card Game Protocol, de https://www.cs.du.edu/~chrisg/publications/pittman-mmsj12.pdf 5 de junio de 2012

[Ajay Yadav] Applied C#.NET Socket Programming de, https://www.c-sharpcorner.com/UploadFile/ajyadav123/applied-C-Sharp-net-socket-programming/ 13 de abril de 2021

[Alan R. Stanger] Unity multiplayer games, Birmingham: Packt Publishing 1st edition, 2013

[Jeff W. Murray] C# game programming cookbook for Unity3D, Boca Raton, Florida: CRC Press 1st edition, 2014

11 Índice de ilustraciones