Creación de un videojuego que combina cartas y ajedrez

Xavier Casadó Benítez

Directora: Mónica Martín Mínguez

Grado: Diseño y desarrollo de videojuegos

Curso: 2023-2024

Universidad: Centre de la Imatge i la Tecnologia Multimèdia - UPC

Índice

Resumen

La aleatoriedad aplicada en los juegos es un elemento que ofrece muchos beneficios, entre los cuales se encuentra la variedad, la cual hace interesante jugar diferentes partidas. Por otro lado, muchos juegos que se enfocan en la competitividad prefieren dejar esta de lado, pues sus efectos pueden suponer varios problemas en el equilibrio del juego.

Este proyecto indaga en las características de la aleatoriedad y su aplicación en juegos, en distintas formas de hacerlo y conceptos de esta y relacionados a tener en cuenta a la hora de plantear el diseño de un juego.

La teoría se llevará a la práctica con la creación de un prototipo de un videojuego desde cero. Este prototipo, llamado Karcha, combinará elementos aleatorios, como puede ser un sistema de cartas y el propio contenido de estas, con un sistema de juego estratégico y competitivo similar al que se puede encontrar en el ajedrez.

Karcha se programará como un juego de 1 contra 1 en línea, desarrollado en Unity y usando herramientas como Steamworks, Fish-Net y Heathen's Toolkit para la conectividad *online* entre jugadores. También se hará uso, entre otras cosas, de inteligencia artificial con el modelo base de Stable Diffusion.

El diseño del juego se iterará en varias ocasiones cuando se encuentren algunos problemas y se ajustará para solventarlos, dirigiéndose a ellos con distintos enfoques.

Finalmente, se evalua el juego mediante sesiones de *playtesting* y otros métodos, en los que se escuchan las opiniones de los jugadores que lo prueban. Los resultados obtenidos apuntan a posibles mejoras de algunos de los elementos del juego, pero aun así son satisfactorios respecto al diseño base.

Palabras clave

Videojuego, Aleatoriedad, Game design, Cartas, Ajedrez, Unity, Steamworks, Fish-Net

Enlaces

Karcha:

https://drive.google.com/file/d/1zxgE5y1AUWlotdAVGTcbpwk9xzMRK5pB/view?usp=sharing

Video:

https://drive.google.com/file/d/1MPmYQExqS92B39tYm4UuURsUDU7gDSb7/view?usp=sharing

Índice de tablas

Índice de figuras

Glosario

API (*Application Programming Interface*):
Funciones y demás recursos que pueden ser usados por los programadores.

balancear:
En el ámbito del diseño de juegos, ajustar elementos del juego (a menudo valores numéricos) con tal de acercarse a un buen equilibrio en su sistema. De esta manera, no hay elementos mucho más poderosos o ventajosos que otros y se considera que el juego es más justo.

ban:
Restricción aplicada a algún usuario que le impide acceder a algunas funcionalidades parciales o totales de un sistema, habitualmente debido a un comportamiento no deseado.

banear:
Acción de aplicar un *ban*.

boss:
En el ámbito de los juegos (y sobre todo videojuegos), se denomina *boss* al enemigo de un juego más poderoso que el resto, con más importancia, ya sea en un nivel o en todo el juego. A menudo es el “líder” de aquellos enemigos menos relevantes.

bug:
Fallo en un software donde algo no funciona como debería.

bullet hell:
Género de videojuegos en el que el jugador debe centrarse en esquivar grandes cantidades de balas enemigas que se desplazan por la pantalla a través del movimiento de su personaje. Además, también es habitual que el jugador pueda disparar sus propias balas.

casual:
Dicho de aquellos juegos que abarcan un público general y amplio, siendo estos accesibles para todo tipo de personas, incluidas aquellas que no están tan acostumbradas a jugar juegos.

cliente:
En el contexto de juegos multijugador P2P, ordenador o dispositivo que participa en el juego y, para ello, se conecta a un servidor.

componente:
En Unity, tipo de *scripts* que se asocia a *game objects*.

de nicho:

Dicho de aquellos juegos que abarcan un público reducido, generalmente grupos de personas con algunas características o gustos en común. No están pensados para un público general o no lo logran alcanzar.

debug:
Proceso de desarrollo de software que consiste en trabajar sobre aspectos del software que tienen — o pueden tener— bugs y/o que tienen que ser ajustados o trabajados mucho más para lograr una versión más estable o definitiva.

diegéticos:
En un juego, elementos que forman parte del mundo o narrativa de este. Un ejemplo es el propio jugador o cualquier otro personaje u objeto.

extradiegéticos:
En un juego, elementos que no forman parte del mundo o narrativa de este. Un ejemplo es, generalmente, la música de fondo o las interfaces en pantalla.

FPS/fps (*Frames Per Second*):
Indicador numérico que indica la cantidad de fotogramas o *frames* que se muestran en pantalla durante un segundo. Por lo general, la cantidad más aceptada como estándar para una experiencia fluida en videojuegos son 60 fps.

framework:
Conjunto de herramientas con una estructura bastante determinada que permite a los desarrolladores de software trabajar en algún aspecto sobre esas bases.

game design (diseño de juego):
Área de trabajo que consiste en crear, analizar y ajustar las experiencias de juego o cualquier característica de su *gameplay*, siendo el objetivo encontrar cohesión y equilibrio entre todos los elementos del juego, además de una buena experiencia para el jugador.

game designer:
Persona que se dedica a estudiar y trabajar en el área del *game design*.

game loop/gameplay loop:
Ciclo de juego en el que el jugador empieza una partida, realiza una serie de acciones (a nivel general, no en detalle) y se termina. También se puede interpretar como el conjunto de esas acciones en sí que se repiten constantemente durante una misma partida.

game object:
En Unity, objeto virtual al cual se le pueden añadir componentes con varias características.

gameplay:

Experiencia de juego que tienen los usuarios mientras juegan. Comprende todas aquellas interacciones que tiene el jugador con el juego, basadas en ciertas normas del mismo.

GDC (Game Developers Conference):
Evento anual muy reconocido en el que diferentes expertos en el ámbito del diseño y el desarrollo de juegos realizan conferencias sobre distintos temas.

host:
En el contexto de juegos multijugador P2P, aquel jugador dentro de un grupo de conexiones al cual el resto se conectan, ya que actúa tanto como cliente como servidor. Generalmente tiene más autoridad que el resto y se puede considerar el "líder" del grupo.

input:
En el contexto de software o videojuegos, aquellas acciones que realiza el usuario para interactuar con el programa. Por ejemplo, mover el ratón de una cierta manera o apretar una tecla específica.

JRPG (Japanese Role-Playing Game):
Género de videojuegos en el que se sigue una estructura típica de combate por turnos, historia y demás elementos propios de los juegos de rol japoneses. En estos, son habituales diferentes estadísticas y habilidades en los personajes.

Lobby:
En el contexto de videojuegos multijugador, sala virtual (a veces abstracta) en la que los jugadores pueden conectarse y mantenerse conectados entre ellos, generalmente antes de empezar una partida. En el contexto de este proyecto, la palabra "sala" puede usarse como sinónimo.

MMO (*Massively Multiplayer Online*):
Género de videojuegos multijugador en el que un gran número de jugadores se conectan en un mismo "mundo". Una de las variantes más populares es el *Massively Multiplayer Online Role-Playing Game* (MMORPG), el cual combina las características típicas de los juegos de rol con esta idea.

multijugador:
Dicho de aquellos juegos en los actúan como jugadores más de una persona. Las interacciones entre los distintos jugadores pueden ser variadas, desde competitivas hasta cooperativas.

open source:
Se denomina así al software de código abierto y público que promueve la colaboración entre usuarios para su desarrollo y su libre uso.

P2P (*Peer to Peer*):
Tipo de conexión entre dispositivos que se basa en ser directa y no depender de un servidor centralizado como intermediario

placeholder:
Elemento provisional usado para transmitir una idea o concepto similar al deseado sin invertir tanto tiempo como requeriría el uso de un elemento finalizado o definitivo.

plataformas:
Género de videojuegos en el que el jugador se enfrenta a retos o enemigos mediante el control de su personaje, siendo el salto de este la principal mecánica a usar para recorrer el escenario desde un punto a otro.

publisher:
Empresa que se asocia con desarrolladores de juegos en su etapa temprana de desarrollo para proporcionarles algunos recursos (como visibilidad, marketing y/o dinero).

rejugabilidad:
Característica de los juegos que, a través de sus elementos, incentiva a los jugadores a jugar varias partidas. Normalmente esto se debe a que hay variación o contenido nuevo en cada una de estas partidas, pese a tratarse del mismo juego.

rejugable:
Dicho de aquellos juegos con una alta rejugabilidad.

RNG (Random Number Generation):
Proceso que consiste en obtener un número aleatorio o impredecible. En el ámbito de los videojuegos y muchos softwares, habitualmente se hace uso de algoritmos matemáticos para lograrlo. Aquellos mecanismos que emplean este proceso son llamados Random Number Generator y reciben las mismas siglas (RNG).

robar:
En el contexto de un juego de cartas, acción de coger una carta del mazo principal (generalmente sin que se sepa previamente cuál es) y, por lo general, añadir-la a la mano del jugador.

roguelike:
Género de videojuegos en el que el jugador recorre escenarios generados proceduralmente y la muerte del personaje que controla es un elemento recurrente en el juego, dando pie a diversas partidas distintas. Aunque estas son algunas de las características más destacables, cabe mencionar que es un género con muchas variaciones y que no siempre sigue la misma estructura o características. Aquellos juegos que difieren más de la definición clásica son comúnmente denominados *rogue-lite*.

run:
Se denomina así a una partida de un juego, normalmente comprende los eventos de juego desde que el personaje del jugador y/o el escenario se crea hasta que se gana o pierde el juego. Normalmente esta palabra es utilizada en juegos donde es habitual hacer muchas partidas; juegos rejugables como los roguelikes.

script:
Archivo con código ejecutable.

servidor:
En el contexto de juegos multijugador P2P, ordenador o dispositivo que coordina y facilita la conexión entre los jugadores.

shooter:
Género de videojuegos en el que el jugador se enfrenta a retos o enemigos mediante armas de larga distancia, normalmente armas de fuego. Es un género popular que a menudo se combina con una cámara en primera persona (First Person Shooter o FPS) y/o multijugador en línea.

Steam:
Plataforma muy popular de distribución digital de software, principalmente videojuegos. Fue desarrollada por Valve Corporation y publicada en 2003. Valve Corporation también ofrece herramientas de desarrollo de software llamadas Steamworks, que ofrecen funcionalidades en juegos que se conectan a Steam.

testear/playtest:
En el ámbito de los juegos, comprobar si el juego funciona correctamente (tanto a nivel de *bugs* como de experiencia del jugador) a través de jugar e interactuar con él, especialmente en etapas tempranas de desarrollo donde no se dispone de *feedback*.

testeo/playtesteo/playtesting:
Sesión o tiempo dedicado a *testear*.

tester/playtester:
Persona que se dedica a *testear*.

toolkit:
Conjunto de herramientas con una estructura más o menos flexible que permite o facilita a los desarrolladores de software trabajar en algún aspecto concreto.

wireframe:
Representación visual simplificada de una interfaz. Sirve sobre todo para planificar el diseño y estructura de la interfaz y sus elementos, sin entrar en detalle en estos.

1. Introducción

La industria de los videojuegos está viviendo un gran crecimiento a nivel global. Esto, en parte, se debe a todas las facilidades que ofrece el medio digital hoy en día, tanto a nivel de producción del juego como de su distribución. Por otro lado, los videojuegos físicos han ido perdiendo relevancia con el paso del tiempo.

Steam es la plataforma de distribución digital de juegos más reconocida. Gracias a algunos de sus datos, es posible hacerse una idea de la magnitud de la industria actual. En el pasado año 2023, un total de 14.531 videojuegos fueron publicados, lo que hace una media de unos 40 juegos que son publicados cada día en la plataforma, aunque esto es sin tener en cuenta que la mayoría de juegos se lanzan en unos periodos de tiempo más estratégicos que otros, por lo que la acumulación de juegos nuevos en ese tiempo es mayor (Way Too Many Games Were Released On Steam In 2023, 2024).

Un factor muy importante a la hora de hacer un juego es destacar por encima del resto en su diseño. Un buen diseño hace que los jugadores tengan una buena experiencia de juego y quieran seguir jugando en futuras sesiones, además de recomendar el juego a sus amigos, en especial si se trata de un juego en línea y permite que estos interactúen entre sí.

Un elemento que se puede incorporar al diseño para alentar esa rejugabilidad y buena experiencia de juego, entre otras cosas, es la aleatoriedad. Sin embargo, esta debe ser incorporada en el diseño de una manera muy cautelosa, pues un mal uso puede causar malas sensaciones para el jugador o incluso un desequilibrio en todo el juego que produce un efecto contrario al que se busca; una mala experiencia. Esto cobra muchísima más importancia al tratarse de juegos competitivos multijugador (normalmente *online*), ya que el beneficio de un jugador perjudica a otro y viceversa.

Este proyecto busca indagar en esta cuestión por medio de la creación de un prototipado de videojuego competitivo multijugador *online* que aplique un diseño en el que la aleatoriedad está involucrada. Se diseñará su sistema de juego e interfaz, se programará su funcionamiento y se aplicará todo el arte y demás elementos necesarios. Por último, se analizarán los resultados y se sacarán conclusiones a partir de eso.

1.1. Motivación

La motivación que me lleva a realizar este proyecto viene de mis ganas por crear un producto que pueda ser útil o valorado por otras personas. Desde hace tiempo he estado pensando en crear un juego que pueda aportar algún tipo de valor o ser interesante para el público y creo que esta es la oportunidad perfecta para poner en práctica mis conocimientos adquiridos en los estudios y llevarlo a cabo. Durante este proyecto planeo ponerme a prueba y ver cómo soy capaz de afrontar los retos que se me presentarán. En el proceso, aprenderé sobre muchos temas relacionados con el diseño, la programación y el arte, y desarrollaré mis habilidades en estos ámbitos. Así pues, creo que puede resultar una experiencia muy beneficiosa con la que saldré más preparado de cara a mis futuros proyectos.

1.2. Formulación del problema

La principal problemática a la hora de aplicar la aleatoriedad en el diseño de un juego es de qué manera hacerlo para mantener un *gameplay* justo y equilibrado, especialmente en juegos competitivos multijugador. Aunque a lo largo del tiempo se han ido creando y definiendo algunos conceptos entorno a esto, por lo general no hay “reglas” o “pautas” muy claras a seguir, sino que en la mayoría de veces toda la responsabilidad del equilibro del juego recae en el *playtesting*. Aunque el playtesting es, sin duda, una gran herramienta a usar, el enfoque de este proyecto será abordar el tema especialmente desde el punto de vista del diseño base del juego. A partir de esto, se intentará llegar a un producto que pueda aportar aspectos interesantes al sector de los videojuegos o de su desarrollo.

1.3. Objetivo general del TFG

El objetivo general del proyecto es crear un prototipo de juego en línea que combine de manera eficiente una estrategia parecida a la del ajedrez con un sistema de cartas, de manera que se mantenga ese elemento competitivo atractivo para algunos jugadores a la vez que hay presente una aleatoriedad llamativa para otros.

1.4. Objetivos específicos del TFG

- Diseñar un sistema de juego entretenido y que ofrezca una buena experiencia.
- Diseñar un sistema de juego equilibrado y justo.
- Diseñar una interfaz de juego intuitiva y atractiva.

- Programar un sistema *online* que funcione correctamente.
- Programar un *gameplay* que funcione correctamente.
- Lograr que el juego se vea pulido y estéticamente bien.

1.5. Alcance del proyecto

El proyecto tiene como meta llegar a un prototipo funcional que cumpla los requisitos especificados anteriormente. Estos objetivos pueden desarrollarse más o menos dependiendo el tiempo que se le dedique a estos, ya que por lo general no hay un único resultado concreto a lograr. Sin embargo, se intentará llegar al resultado más óptimo posible teniendo en consideración el tiempo límite del proyecto.

Además, la creación del prototipo no tiene por qué ser necesariamente el final. Si el resultado es lo suficientemente conveniente, el prototipo puede llegar a consolidarse como un videojuego más dentro del mercado en el futuro.

Por otra parte, no está de más tener presente que el tiempo límite mencionado es el principal factor que puede limitar el proyecto o suponer un riesgo para este en caso de resultar escaso. Para evitar esto, se planificarán con antelación los distintos pasos a seguir en el desarrollo y no se descartará el “adaptar” estos pasos durante el desarrollo para reducir la carga de trabajo en caso de ser necesario.

En cuanto al público, el prototipo se enfoca en un público muy general y amplio. Así como el ajedrez, incluye todas las edades, aunque sería más recomendado su uso a partir de los 5 o 7 años considerando la complejidad del juego. Busca ser un juego bastante casual para abarcar la mayor cantidad de personas posible. Aquellos que, concretamente, pueden estar más interesados en el juego son aquellas personas que tienen experiencia o buscan iniciarse en juegos de estrategia, cartas y/o competición *online*.

Por último, los resultados obtenidos del proyecto pueden ser beneficiosos para la comunidad de desarrolladores de juegos, tanto a nivel de diseño, como de programación como de arte. Varios aspectos del proyecto se podrán usar como una referencia en futuros proyectos, ya sea en análisis, comparaciones, ejemplos, etc.

2. Estado del arte

2.1. Marco teórico

2.1.1. Propiedades de un juego

Hay una gran cantidad de definiciones de lo que es un “juego” propuestas por diferentes pensadores a lo largo de la historia. Muchas de estas comparten algunos puntos en común, pero hay otras características en las que difieren, por lo que ninguna está consolidada como la más correcta o definitiva. Por otra parte, muchos de estos pensadores han dejado su huella y propuesto teorías y conceptos muy interesantes acerca de los juegos y la diversión, siendo Roger Caillois uno de los más aclamados.

Roger Caillois publicó en 1958 un libro titulado *Les jeux et les Hommes* (Caillois, 1958), traducido al inglés como *Man, Play and Games* en 1961 por Meyer Barash (Caillois & Barash, 2001). En este, Caillois discernía dos clasificaciones para los juegos. En la primera, separaba estos según si eran *Paidia* (juegos con normas no muy estrictas, flexibles, más dados a la improvisación y enfocados en divertirse en el sentido más puro de la palabra) o *Ludus* (juegos con una estructura y reglas más fijas e invariables, las cuales los jugadores tienen que respetar y usar junto a su habilidad, esfuerzo, destreza, etc.).

En la segunda clasificación, distinguía cuatro principales características en las que se basaban los juegos. Estas eran:

— *Agon*: Se enfocan en la competencia entre jugadores, donde estos se enfrentan en igualdad de condiciones y se pretende demostrar quién es más hábil o superior en unas condiciones ideales y justas para ambos.

— *Alea*: Se centran en el azar. El juego se rige por esta en gran medida, por lo que se decanta por el jugador más afortunado, el que tiene más suerte ante un escenario igual para todos.

— *Mimicry*: Se caracteriza por la interpretación, el simular otra realidad diferente e imaginarse dentro de esta. La intención es mantener esa ilusión durante todo el juego, en donde se cree estar en una situación completamente ficticia.

— *Ilinx*: Se especializan en romper la estabilidad de percepción de los jugadores. Buscan llevarlos a algún extremo como lo es el pánico, el vértigo o el riesgo, haciendo que se desprenda adrenalina y se exploren sensaciones.

(Emil_lab, 2024)

Al seguir una serie de normas preestablecidas casi inevitablemente, prácticamente todos los juegos de mesa y videojuegos pertenecen al grupo de juegos *Ludus*, sean estas reglas más o menos estrictas. Sin embargo, respecto a la segunda clasificación, los resultados son muy variados, ya que no tan solo los juegos pueden decantarse por cualquiera de los aspectos de la clasificación, sino que es prácticamente inevitable en muchos de ellos centrarse exclusivamente en uno solo de ellos. La mayoría de juegos abarcan todas las propiedades en mayor o menor medida, variando los porcentajes según el juego y lo que busque transmitir.

Algo curioso, sin embargo, es el hecho de que en muchos juegos coexistan elementos *Agon* con elementos *Alea*, ya que estos representan valores totalmente opuestos el uno del otro. Esta combinación, si bien es cierto que le proporciona variedad y complejidad al juego, puede llevar fácilmente a un desequilibrio del mismo. Esto es, dándole una importancia excesiva a la aleatoriedad en un juego enfocado a la destreza de los jugadores o, por el contrario, exigiendo una habilidad demasiado alta en un juego en donde los jugadores no buscan eso.

2.1.2. Definición de la aleatoriedad

Cabe aclarar que el concepto de aleatoriedad, en el contexto en el que va a usarse de aquí en adelante, no es del todo real. Un dado realmente no obtiene un número aleatorio de la nada, sino que sigue las leyes de la física y lo obtiene a partir de una información muy amplia como puede ser su masa, la fuerza con la que se tira, la trayectoria del lanzamiento, el rozamiento con la superficie, la fuerza del viento, etc.

De igual manera, un ordenador sigue los mismos principios y no es capaz de generar números aleatorios de la nada, sino que lo hace a través de algoritmos matemáticos que usan como base algún número con mucha variabilidad, dependiendo de la situación. Ese número, de nuevo, no se “crea” sino que se obtiene del entorno, ya sea a partir de la temperatura de los componentes del ordenador, la hora actual, el número de *FPS* del programa o interacciones del usuario.

Estas interacciones o *inputs* son usados a menudo para generar una aparente aleatoriedad en videojuegos. Algunos ejemplos son los píxeles que recorre el ratón, el tiempo que se tarda en realizar alguna acción o información de la partida como cantidades de objetos, puntos de vida de personajes, niveles, número de pasos que da el personaje, etc.

Debido a que toda esta información es demasiado compleja para ser considerada por los humanos en predicciones, el resultado se puede considerar aleatorio, a pesar de que realmente se trata de una *pseudo-aleatoriedad*.

Por otra parte, no está de más señalar que una aleatoriedad más "real" es aquella que se obtiene de los entornos más desconocidos y menos entendidos por los humanos. Así pues, aquellas simulaciones que necesitan una aleatoriedad más fiable a menudo usan información como el ruido cósmico o comportamientos subatómicos o cuánticos. Sin embargo, esto no concierne al mundo del juego por ser demasiado costoso e innecesario (Randomness in Games: A Long-form Analysis - YouTube, s. f.).

Dicho esto, la aleatoriedad puede llegar a ser un elemento muy interesante y valioso cuando se aplica a los juegos. Es algo que crea interés en los jugadores, incertidumbre en lo que sucederá, y es una buena manera tanto de llamar su atención como de mantenerlos activos en el juego, además de, en algunos casos, incluso llegar a aprovecharse para extender las posibilidades de un juego hasta el infinito (siendo este el caso de juegos con generación procedural como *Minecraft*[1], en el que se genera un mundo nuevo aleatorio en cada partida) (Aguilla, 2022).

Además, teniendo en cuenta los avances tecnológicos más recientes como la incorporación de la inteligencia artificial a diferentes procedimientos de creación, no resulta descabellado pensar que es un recurso que se explorará y aprovechará enormemente en el futuro, entre otras cosas para la creación de contenido para juegos.

Por otro lado, la aleatoriedad pura es algo que, de por sí, escapa al jugador. Este no tiene control sobre esta y sus resultados, lo que puede llevar en muchos casos al efecto contrario que se busca: una frustración del jugador, que sienta impotencia o que incluso

[1] Videojuego *sandbox* desarrollado por Mojang Studios y publicado en 2011.

se sienta mal por su suerte o el trato que le da el juego, pudiéndose considerar en algunos casos un trato injusto.

Así pues, muchos juegos han intentado aprovecharse al máximo de los beneficios de la aleatoriedad, pero algunos no han podido evitar caer en sus efectos negativos. Esto puede deberse tanto al planteamiento inicial del sistema de juego, en donde no existe una buena relación base entre el *Agon* y el *Alea*, como en un mal uso de la *Alea* solamente, en donde esta tiene más o menos peso del que debería de tener para contentar a los jugadores.

2.1.3. La aplicación de la aleatoriedad en juegos

Aunque la aleatoriedad se puede implementar de diferentes maneras dependiendo del juego, siempre se debe tener en cuenta en qué afecta al juego exactamente y si su adición permite lograr lo que se busca de manera armónica con todo el sistema de juego.

“Un desarrollador siempre debería valorar qué aporta a su juego una mecánica aleatoria.”
(de 3DJuegos, 2021)

Los principales motivos por los que se aplica la aleatoriedad en los juegos son los siguientes:

Variedad

La aleatoriedad permite crear variedad de una manera muy fácil y replicable, en especial si se tienen en cuenta, no simplemente los elementos aleatorios individuales en sí, sino las diferentes combinaciones entre estos, llegando a poder recrear infinitos escenarios, niveles, personajes y problemas. Gracias a esto, se puede hacer que cada partida empiece o se desarrolle de una manera distinta, lo que le da frescor al juego y lo hace menos repetitivo para el jugador. Además, la aleatoriedad hace que el jugador esté mucho más interesado en ver cómo se desarrolla el juego, pues es impredecible.

A pesar de esto, cabe resaltar que, en caso de buscar variedad en el juego a través de una generación aleatoria de contenido (por ejemplo, que se genere un mapa aleatorio en cada partida), dicha variedad destacará siempre, por lo general, por su cantidad y no por su calidad; pues algo diseñado y creado a mano, enfocado a algo concreto, suele tener mejor calidad que algo generado artificialmente con un propósito más general.

Por último, otro aspecto a tener en cuenta es el hecho de que, en el caso de que los niveles sean aleatorios y varíen entre partidas, esto imposibilita que los jugadores se adapten a los niveles específicos, porque estos serán diferentes cada vez. En su lugar, los jugadores se verán forzados a aprender a jugar mejor en general, en cualquier circunstancia, lo que por lo general es lo que se pretende que haga el jugador.

Emoción

Desde el punto de vista de las emociones que le transmite el juego al jugador, una aleatoriedad bien implementada puede ser muy enriquecedora.

La impredecibilidad de la aleatoriedad supone que se pueden dar casos muy poco comunes en el juego que provoquen emociones fuertes en los jugadores como lo es la sorpresa (por ejemplo que un jugador obtenga un objeto muy poco común o que realice un ataque crítico en el momento más oportuno). Estos casos no solamente emocionan a los jugadores una vez aparecen, sino también antes de hacerlo (o incluso si no llegan a aparecer). Por ejemplo, si el jugador espera con ansias que el enemigo falle su ataque, pues es su única forma de sobrevivir. Del mismo modo, los jugadores también sienten más emoción cuando quieren conseguir una recompensa aleatoria en vez de una que ya conocen.

Por otra parte, también se ha de recalcar que un mal uso de la aleatoriedad puede llevar fácilmente a la frustración del jugador cuando esta no le beneficia, la cual es una emoción negativa.

Equilibrio

La aleatoriedad puede dar ventaja a jugadores menos habilidosos ocasionalmente. En juegos multijugador, la aleatoriedad le resta importancia a la habilidad de los jugadores, lo que permite que, en algunas situaciones, jugadores nuevos o más inexpertos puedan tener alguna oportunidad frente a jugadores más expertos, lo que hace que el resultado no esté decidido desde el primer momento de la partida. Esto hace que el menos habilidoso se mantenga con optimismo y motivación, mientras que el más habilidoso no puede relajarse.

Una buena praxis sería que esto se aplicara más en jugadores más inexpertos, pero cada vez menos a medida que el nivel de los jugadores es mayor. Sería ideal que, de alguna manera, cuanto más experimentados sean los jugadores, la aleatoriedad del

juego influya menos en estos. Por ejemplo, en un juego donde los jugadores obtienen armas aleatorias, el tipo de arma puede significar una ventaja para uno u otro de los jugadores al enfrentarse, pero cuanto mejor sepan los jugadores usar las mecánicas del juego, ya sea por destreza (apuntar bien, construir coberturas, esquivar, bloquear, hacer combos, etc.) o conocimiento (saber en qué momento esprintar, en qué situación atacar o mantener el sigilo, predecir y contrarrestar al rival, etc.) el tipo de arma será un elemento menos decisivo.

Por lo general, muchos game designers opinan que un juego enfocado en el competitivo no debería tener mucha aleatoriedad, mientras que cuanto más casual sea y quiera recoger a un público más amplio con diferentes niveles de destreza y experiencia, más aleatoriedad debería tener o, por lo menos, esta estaría más justificada.

En este caso, la dificultad radicaría en crear un juego entre medias de estos dos extremos. Dependiendo la cantidad de *Alea* y *Agon* del sistema de juego y su uso, el rango de público podrá llegar a ser más amplio o más reducido, y se decantará más hacia jugadores que juegan por pura diversión o los que lo hacen por competición, respectivamente.

Complejidad

La aleatoriedad en muchas ocasiones obliga a los jugadores a adaptarse a circunstancias inesperadas que puedan ser provocadas por esta. Así pues, los jugadores deben hacer estrategias más amplias que contemplen la mayor cantidad de resultados posibles, en muchas ocasiones teniendo que decidir entre algo con más riesgo pero mejores resultados o lo opuesto. Esto también incluye diferentes planes de contención que los jugadores deben planificar por si no son afortunados o, por otra parte, replantearse nuevas estrategias cuando esto sucede.

(de 3DJuegos, 2021; *The Two Types of Random in Game Design - YouTube*, s. f.)

2.1.4. El flow de información

Así pues, la aleatoriedad impide que los jugadores puedan planear estrategias sabiendo toda la información, lo que le añade complejidad a la experiencia de juego y muchas veces replanteamientos de la estrategia por parte del jugador, haciendo que se tenga que adaptar a la nueva información que le llega. Es importante tener en consideración cada cuanto tiempo se aplicará la aleatoriedad durante el juego. Esto representa lo que

Ethan Hoeppner definió como *flow de información* en su artículo *Plan Disruption* (Hoeppner, s. f.) y supone la frecuencia en la que los jugadores deciden si deben seguir con su estrategia, adaptarse y replantearla o cambiar a una distinta (The Two Types of Random in Game Design - YouTube, s. f.).

Figura 1. Gráfico representativo de un buen flow de información

(The Two Types of Random in Game Design - YouTube, s. f.)

Este gráfico (*ver Figura 1*) muestra un buen ejemplo de cómo aplicar nueva información (aleatoriedad) durante el juego y cómo debe afectar esta a los planes del jugador, de manera que se cree un buen *flow* de información que respete su estrategia por lo general, pero que le obliga a adaptarse a nuevas condiciones cada cierto tiempo.

2.1.5. El horizonte de información

Cada vez que el jugador recibe información nueva, este replanteará toda la situación desde el punto actual en el que se encuentra, así como lo hace un jugador de ajedrez cada turno. El jugador, sin embargo, tiene un límite a la hora de visualizar toda la información del juego, ya sea en ese mismo turno o imaginando posibles jugadas consecutivas y situaciones de turnos futuros. Este límite puede estar delimitado por las propias capacidades del jugador (un jugador de ajedrez visualizando las próximas posibles jugadas de la partida) o por el propio sistema del juego, debido a sus normas (la niebla de guerra de *StarCraft*[2], la cual impide al jugador ver más allá del terreno que conoce). Este límite fue nombrado por Keith Burgun como *el horizonte de información*.

[2] Saga de videojuego de estrategia en tiempo real (*RTS*) desarrollada por Blizzard Entertainment.

Según él, los *game designers* deben encargarse de establecer este horizonte correctamente. Deben encontrar un equilibrio entre "situar" el horizonte de información los suficientemente "cerca" como para que el juego no sea una competición sobre "mirar más lejos" y lo suficientemente "lejos" como para que los jugadores tengan el suficiente margen para planificar una estrategia y que el juego no sea demasiado aleatorio.

(«Uncapped Look-Ahead and the Information Horizon», 2014)

"It's a very difficult balancing act. If the information horizon is even a little bit too close, the game becomes too random. If it's too far out, you risk having a look-ahead contest problem. You must find the right balance, and have a system that is rich and interesting enough to allow for creative play within a restricted space."

(«Uncapped Look-Ahead and the Information Horizon», 2014)

Para delimitar el horizonte de información, hay que poner unos límites a la información del juego a la que el jugador tiene acceso; hay que determinar qué información debe ser inaccesible para este. Un tipo de información inaccesible es la aleatoriedad anteriormente mencionada. Sin embargo, hay otros tipos de información inaccesible. Todos ellos se pueden agrupar bajo el concepto de "incertidumbre".

2.1.6. La incertidumbre

Roger Caillois ya habló de la incertidumbre en su libro *Man, Play and Games* (Caillois & Barash, 2001), en donde la destacó como algo esencial en un juego. Sin esta, los juegos simplemente no serían divertidos y las personas no tendrían motivos para jugar, así como les pasa a las personas muy habilidosas que ganan sin dificultad.

"Doubt must remain until the end, and hinges upon the denouement. In a card game, when the outcome is no longer in doubt, play stops and the players lay down their hands. [...] Every game of skill, by definition, involves the risk for the player of missing his stroke, and the threat of defeat, without which the game would no longer be pleasing. In fact, the game is no longer pleasing to one who, because he is too well trained or skillful, wins effortlessly and infallibly."

(Caillois & Barash, 2001)

A partir de esto, Greg Costikyan es un *game designer* que profundizó bastante en este tema. Él considera que, al contrario de lo que decía Roger Caillois, un juego no

necesariamente depende del resultado final (por ejemplo, en *Space Invaders*[3] el jugador sabe que acabará perdiendo en algún momento; además, muchos juegos MMO no tienen un final claro). Sin embargo, sí que apoya su idea general respecto a la incertidumbre, y lo refuerza poniendo como ejemplo el juego *Tic-Tac-Toe*[4], el cual es divertido mientras los jugadores no sepan las estrategias vencedoras, pero una vez lo hacen, si siempre usan las mismas jugadas óptimas y llegan al mismo resultado, pierde su sentido.

Así pues, Greg Costikyan complementa este concepto y define diferentes tipos de incertidumbre:

Performative uncertainty

La incertidumbre radica en si el jugador será capaz, mecánicamente, de realizar las acciones que se plantea. Por ejemplo, si el jugador será capaz de hacer un salto en el momento correcto en *Super Mario Bros*[5].

Solver uncertainty

La incertidumbre radica en si el jugador será capaz, a nivel cognitivo o mental, de llegar a las conclusiones correctas que le permitan avanzar. Por ejemplo, si el jugador será capaz de solucionar los acertijos de *The Secret of Monkey Island*[6].

Player uncertainty

La incertidumbre radica en tratar con otras personas y no saber cuáles serán sus acciones. Por ejemplo, no saber a qué lado se moverá un jugador enemigo en *Team Fortress 2*[7].

[3] Videojuego clásico *shoot 'em up* desarrollado por Taito y publicado en 1978 para máquinas arcade.
[4] Juego popular que consiste en marcar 3 cruces o círculos en línea antes que el oponente, en un tablero de *3x3* casillas. También se conoce como ceros y cruces o tres en línea/raya, entre otros.
[5] Conocido videojuego *plataformas 2D* desarrollado por Nintendo y publicado en 1985.
[6] Videojuego de aventura gráfica *point-and-click* desarrollado por Lucasfilm Games y publicado en 1990.
[7] Videojuego *FPS* multijugador *online* desarrollado por Valve Corporation y publicado en 2007.

Analytic uncertainty

La incertidumbre radica en “ver” o “analizar” una situación que está cada vez más lejos del punto inicial y supone una mayor complejidad por las posibilidades que se expanden. Por ejemplo, ver lo que puede pasar en el tercer turno a futuro a partir del turno actual en una partida de ajedrez. En este caso, el juego tiene una gran capacidad de ayudar al jugador a lidiar con esta incertidumbre, por ejemplo con ayudas visuales; o incluso incrementar la dificultad que supone, por ejemplo omitiendo información.

Randomness uncertainty

La incertidumbre radica en los elementos aleatorios de un juego. Por ejemplo, los efectos de equiparse un objeto en *NetHack*[8].

Hidden information uncertainty

La incertidumbre radica en aquellos elementos del juego que no le han sido revelados al jugador, lo que muchas veces complementa a aquellos que sí le han sido revelados. Por ejemplo, las cartas de la mano de un rival en el Póker.

Schedule uncertainty

La incertidumbre radica en el intento de los jugadores por combinar el juego con su propia vida u horario. Por ejemplo, dejar de jugar a *CityVille*[9] y volver a jugar dentro de 30 minutos para recolectar recompensas de una manera óptima.

Narrative uncertainty

La incertidumbre radica en la historia o narrativa del juego, en que el jugador no sepa cómo se desarrollará ésta.

Development anticipation

La incertidumbre radica en los nuevos elementos que se han agregado al juego en una actualización o adición de contenido.

[8] Videojuego *roguelike* desarrollado por The NetHack DevTeam y publicado en 1987.
[9] Videojuego *city-building* desarrollado por Zynga y publicado en 2010 para Facebook.

Uncertainty of perception

La incertidumbre radica en que los jugadores obtengan una visión correcta del juego y de su escenario. Por ejemplo, que el jugador pueda encontrar un objeto oculto en *Mystery Case Files: Huntsville*[10]. Otro ejemplo es que el jugador distinga qué balas son peligrosas para él en un juego *bullet hell* donde la pantalla está repleta de elementos y efectos visuales.

Semiotic uncertainty

Hace referencia al concepto *semiotic contingency* de Thomas M. Malaby en su artículo *Beyond Play: A New Approach to Games* (2007). La incertidumbre radica en la interpretación que el usuario le da a lo que sucede en el juego. Por ejemplo, que el jugador asocie tocar una bandera con ganar en *Eryi's Action*[11] por sus similitudes con *Super Mario Bros.*, aunque realmente esa acción haga que el jugador pierda. Otro ejemplo es cuando en el juego de mesa *Train*[12] se desvela al final que todos los pasajeros transportados gracias a los jugadores se dirigen a un campo de concentración nazi.

(*«Uncertainty in Games» Greg Costikyan, Playdom - YouTube*, s. f.)

Todos estos tipos de incertidumbre limitan la experiencia del jugador y/o la información de la que este dispone y deben tenerse en cuenta a la hora de plantear el horizonte de información de los jugadores. En relación a esto, existe un concepto importante a la hora de hacer un juego que ofrezca muchas posibilidades al jugador, en especial si se trata de un juego con mucho *analytic uncertainty*; la parálisis por análisis.

Se trata de un estado en el que la persona es incapaz de realizar ninguna acción por estar abrumada por todas las opciones que tiene y/o la complejidad o número de elementos implicados en la decisión («Analysis Paralysis», 2024). Es importante para un buen flujo y ritmo de juego que los jugadores no caigan en este estado que les impide tomar acciones. Para ello, debe controlarse y en muchos casos limitarse la cantidad y/o complejidad de la información de la que los jugadores disponen, ya sea ocultando información directamente (*hidden information uncertainty*) o modificando otros tipos de

[10] Videojuego de puzles tipo *hidden object* desarrollado por Big Fish Games y publicado en 2005.
[11] Videojuego plataformas 2D desarrollado por Xtal Sword y publicado en 2011.
[12] Juego de mesa diseñado por Brenda Romero y publicado en 2009.

incertidumbre del juego, como la aleatoriedad (*randomness uncertainty*) para hacerlo más simple.

2.1.7. Tipos de aleatoriedad

Por otro lado, la aleatoriedad en sí misma puede clasificarse en diferentes tipos según los *game designers*. Los dos más conocidos son *input randomness* y *output randomness*.

Estos términos fueron nombrados por Keith Burgun en su artículo *Randomness and Game Design* (2014). Por otra parte, también son conocidos como *Pre-Luck* and *Post-Luck*, que es la manera en la que anteriormente Soren Johnson los mencionó en su conferencia de la GDC 2014, ese mismo año (A Study in Transparency, s. f.).

— ***Input randomness*:**

  Plantea una situación al jugador con información nueva, y el jugador debe tomar acciones sobre esta. Apoya la estrategia, ya que el jugador debe planificar sus acciones futuras en base el escenario creado por el *input randomness*. Ejemplos de esta son la generación de mapas, robar cartas al inicio de turno u obtener ciertos recursos para utilizar.

— ***Output randomness*:**

  Decide los resultados de los eventos de forma aleatoria, lo que hace que no todo salga como el jugador lo plantea inicialmente. Debilita la estrategia, ya que el jugador debe optar por estrategias más amplias y generales, menos sólidas (o crear planes de contingencia) que tengan en cuenta diferentes resoluciones de las *output randomness*, o por otro lado usar la improvisación. Ejemplos de esta son la probabilidad de acertar una acción o de que el ataque de un enemigo falle.

Cabe resaltar que, aunque estos dos términos muchas veces se polarizan, no son algo totalmente binario, sino que hay un espectro entre ellos.

Por una parte, en muchas ocasiones el *output randomness* se convierte en *input randomness* de cara al siguiente turno o fase del juego, ya que los resultados de una fase definen el escenario en el que inicia la siguiente.

Por la otra, una mala implementación de *input randomness* puede provocar problemas muy parecidos a los propios del *output randomness*, ya que el jugador no podría beneficiarse adecuadamente de los beneficios del *input randomness* y la aleatoriedad pasaría a estar más descontrolada para el jugador, como ocurre con el *output randomness*. Es decir, cuanta menos capacidad —o tiempo— de reacción se le dé al jugador, la aleatoriedad pasará a ser más *output randomness* que *input randomness*.

“Interestingly, while these two types are certainly distinct enough from each other to warrant the classifications, they do technically exist on a continuum. Without going into much detail on it, it should be noted that irresponsible use of input randomness – where the player has very little time to respond to the new information, or where the game generates problems of wildly varying difficulty match to match – causes similar problems as output randomness.”

(«Randomness and Game Design», 2014)

“Input randomness, when put up close enough to the player so that he can’t plan around it, is basically output randomness.”

(«Randomness and Game Design», 2014)

Cabe considerar que el output *randomness* puede usarse en algunos juegos para simular fallos en las ejecuciones de algunas acciones. En este caso, haciéndose referencia a acciones que el jugador decide llevar a cabo de una manera teórica (como elegir que un personaje ataque a otro en un juego JRPG), en contraparte a acciones que el jugador debe realizar de forma práctica usando su habilidad (como disparar a un jugador en un juego *shooter* o saltar un agujero en un juego *plataformas*) y que, en algunos casos, fallarán en la ejecución por un simple fallo mecánico (al igual que puede fallar un ataque “teórico” de un JRPG).

(«Randomness and Game Design», 2014; Zhang et al., 2021)

Otra distinción que añade Keith Burgun a la aleatoriedad es el de la aleatoriedad variable y la uniforme.

La aleatoriedad variable es aquella que no tiene consideración por ajustar mínimamente su rango de aleatoriedad y resultados que pueden darse a la largo del juego. Es la que no está controlada para evitar resultados excesivamente buenos o excesivamente malos, tanto a nivel de eventos aleatorios individuales (como robar una carta que puede ser excesivamente buena o excesivamente mala) como en una serie de eventos

aleatorios en conjunto (como que, en Tetris[13], al jugador le tocase una misma pieza durante 4 turnos seguidos).

En la otra cara de la moneda, la aleatoriedad uniforme es aquella que pone ciertas limitaciones a la aleatoriedad con tal de controlarla en cierta forma. Cuanto más uniforme sea la aleatoriedad, significa que sus resultados estarán más restringidos, de nuevo, tanto en casos de eventos particulares como en un conjunto de estos. De este modo, los resultados, aún y ser aleatorios, tienen en cuenta información del juego y se adaptan a ella.

(«Three Types of Bad Randomness, and One Good One», 2018)

A continuación se encuentran algunos ejemplos de aleatoriedad uniforme en algunos juegos.

Para eventos individuales, donde se controla el contenido:

***Diablo III*[14]:** Los objetos que obtienes como recompensas tienen más posibilidades de ser aquellos que pertenecen a la clase de personaje con la que está jugando el jugador en ese momento. De esta manera el jugador recibe recompensas aleatorias pero mayormente útiles.

***Spelunky (2012)*[15]:** Por lo general, los niveles pueden aparecer de manera aleatoria con variaciones, como que el nivel esté oscuro y sea más complicado. Sin embargo, si el jugador completa un nivel en un tiempo corto (20 segundos), esta variante de nivel oscuro no se aplicará al siguiente nivel. De esta manera, si el jugador tarda en completar el nivel, la aleatoriedad puede castigarlo, pero si es rápido, se le recompensará eliminando esa aleatoriedad.

Para conjuntos de eventos, donde se controla el flujo:

[13] Videojuego clásico de puzles creado por Alexey Pajitnov y publicado en 1885.
[14] Videojuego *hack-and-slash* desarrollado por Blizzard Entertainment y publicado en 2012.
[15] Videojuego *rogue-lite* de plataformas desarrollado principalmente por Derek Yu y publicado en 2012. Es una versión mejorada de *Spelunky Classic* (2008).

***Pandemic*[16]:** El mazo de cartas totales se divide en 4 secciones y en cada una de ellas se añade una carta de Epidemia, por lo que estas cartas quedan repartidas a lo largo de todo el mazo.

***Tetris* (algunas ediciones):** No generan los bloques venideros de uno en uno, sino que los generan en paquetes en donde hay todos los diferentes tipos de bloque sin repetirse, en un orden de aparición aleatorio.

(*The Two Types of Random in Game Design - YouTube*, s. f.)

Así pues, es común el uso de varias técnicas como estas que usan los juegos para limitar la aleatoriedad o controlarla en cierto modo, ya que una aleatoriedad total puede, en muchos casos, desestabilizar el juego de una forma descontrolada, tanto por un problema de "contenido" como de "flujo".

Finalmente, la tabla a continuación (*ver Tabla 1*) muestra la clasificación final de la aleatoriedad en los juegos según Keith Burgun, siendo el *uniform input* el tipo de aleatoriedad que él considera mejor y más justo para aplicar en un juego, desde el punto de vista del diseño (por permitir a los jugadores construir estrategias sólidas —*input*— y por adaptar la aleatoriedad y evitar que se descontrole —*uniform*—).

	VARIABLE	UNIFORM
Output Randomness	VARIABLE OUTPUT	UNIFORM OUTPUT
Input Randomness	VARIABLE INPUT	UNIFORM INPUT

Tabla 1. Tipos de aleatoriedad según Keith Burgun

(«Three Types of Bad Randomness, and One Good One», 2018)

[16] Juego de cartas cooperativo diseñado por Matt Leacock y publicado en 2008.

2.1.8. Características de la aleatoriedad

Con tal de llegar a un buen punto cuando se diseña una acción aleatoria en un juego, hay diferentes conceptos deben tenerse en cuenta. Estos conceptos o variables intentan definir cómo es la aleatoriedad de la acción[17] y pueden ajustarse para balancearla.

Delta of randomness

La diferencia que hay entre el mejor resultado aleatorio posible y el peor. En juegos de cartas, la delta se suele aplicar en relación a la *power curve*[18] del juego, para comparar cómo de beneficioso puede ser el mejor y peor resultado aleatorio respecto al promedio. Un resultado negativo suele estar por debajo y uno positivo por encima. Cuanto más arriba de la curva está el mejor resultado (o más probable sea este de suceder), más abajo debería estar el peor para compensarlo. Hacer un delta muy amplio (es decir, con un resultado positivo muy arriba y uno negativo muy abajo) implica que esta decisión aleatoria tiene mucha importancia en el juego, lo que normalmente se pretende evitar si no se quiere que la partida gire en torno a resultados aleatorios y ajenos a los jugadores.

Un ejemplo de un delta amplio sería tirar una moneda para decidir si un jugador pierde o gana. Por el contrario, un delta bajo provoca pequeños momentos emocionantes e inesperados en el juego que mejoran su experiencia, ya que no determina el resultado final de la partida o juego de una forma tan directa.

(*Dissection of Randomness in Games*, s. f.; *The Delta of Randomness - Can You Balance for RNG? - Extra Credits - YouTube*, s. f.)

Randomness information delay

El tiempo o capacidad que tiene un jugador para poder reaccionar ante un evento aleatorio y poder crear una estrategia de juego en torno a su resultado. Cuanta más capacidad de reacción otorgue el evento, más se acercará a ser *input randomness*, mientras que cuanta menos, se acercará más a un *output randomness*.

[17] Entiéndase como acción o evento aleatorio cualquier elemento del juego que afecta a una partida y que involucra cierto grado de aleatoriedad en su resultado.
[18] Gráfico que define el poder promedio de las cartas en función de su coste. *Para más información ver el punto* 2.1.11. El equilibrio de las cartas.

Randomness complexity (o scope)

Medidor de cuantas opciones pueden ocurrir en un evento aleatorio. Por ejemplo, cuántas cartas diferentes pueden salir si robas una carta de un mazo. En este punto también hay que plantearse si el jugador es capaz de plantearse todas esas opciones o hasta qué punto puede hacerlo.

Expected value

Relacionado, el valor que es esperado de un evento aleatorio por parte del jugador, como una “media” o “balance” de lo que puede suceder en una situación habitual, puede ser algo numérico fácilmente evaluable como un rango de puntos entre los que espera obtener un resultado o algo más abstracto como la obtención de un personaje aleatorio o entrar a una sala aleatoria.

Influenciability

Capacidad de los eventos aleatorios para que el jugador pueda moldear su aleatoriedad y adaptar las probabilidades de sus resultados.

- **Sin influenciabilidad:**

  Los jugadores no tienen forma de modificar la aleatoriedad. Por ejemplo, una tirada de dados.

- **Influenciabilidad intrínseca:**

  La generación aleatoria tiene en cuenta elementos del juego que usa en sus algoritmos (por ejemplo la cantidad y orden de objetos, niveles de personajes, puntos de salud, Inputs del jugador, etc.). Así pues, los jugadores, si llegan a descubrir cómo funciona exactamente este proceso, pueden modificar esos elementos de juego con el fin de manipular la aleatoriedad según su voluntad, de forma similar a como un Hacker modificaría valores del código del juego para alterarlo.

- **Influenciabilidad extrínseca:**

  La generación aleatoria tiene en cuenta elementos del juego que, si bien es cierto que no siempre le son indicados al jugador de una manera directa, no pretenden estar ocultos de este. Estos elementos, a diferencia de los que se usan en la

> influenciabilidad intrínseca, no son herramientas usadas por el algoritmo del RNG en su código, sino que son elementos con los que la propia acción aleatoria está relacionada en el juego. Por ejemplo, si un hechizo afecta a un enemigo aleatorio y el jugador quiere que le afecte a un enemigo "X" en concreto, puede reducir el número de enemigos antes de lanzar el hechizo para asegurarse de que este afectará a "X" y no a otro, o por lo menos aumentar las probabilidades de que esto suceda.

(*Dissection of Randomness in Games*, s. f.)

2.1.9. Recursos al implementar aleatoriedad

Como se muestra la aleatoriedad

Siempre que se trata con aleatoriedad, es importante que el jugador tenga en mente una aproximación de las probabilidades que tienen los diferentes resultados posibles, con tal de que el juego sea honesto y transparente con el jugador (The Two Types of Random in Game Design - YouTube, s. f.). De esta forma, el jugador es consciente de los riesgos y beneficios. Esto se lleva a cabo eficientemente mostrando los porcentajes de resultados directamente, aunque según la naturaleza del juego, se puede preferir mantenerlo más encriptado o sencillo usando palabras o iconos en vez de números ("poco probable", "frecuentemente", etc.). Por otra parte, también hay situaciones en las que se pretende que el jugador lo experimente por él mismo y no a través de un texto o icono visual, por lo que este lo aprende mediante su experiencia de juego, a partir de repetir estos eventos aleatorios.

Compensaciones

Algo a tener en cuenta a la hora de decidir el riesgo que supone una acción aleatoria su peso, es que, con tal de que el jugador no se frustre excesivamente al no conseguir un resultado deseado, puede obtener en su lugar un resultado que, si bien no es igual, lo compense en cierto modo (The Two Types of Random in Game Design - YouTube, s. f.). Consiste en no basar acciones aleatorias en un blanco o un negro solamente, sino añadir posibles grises. Un ejemplo sería un ataque que tiene un 90% de probabilidades de acertar y hace 100 puntos de daño, pero que si no llegase a acertar, en vez de fallar completamente y quedar como una acción desperdiciada, que haga 10 puntos de daño en su lugar, o que exista esa posibilidad.

Porcentajes

Un recurso que se usa en algunos juegos consiste en mostrar porcentajes en pantalla que son ligeramente diferentes a los reales, siendo los reales más beneficiosos para el jugador que los mostrados. Esto se lleva a cabo para evitar frustración en los jugadores, que puede llegar a ocurrir en caso de que un resultado deseado y con altas probabilidades de salir no lo haga. Así pues, se le hace creer al jugador que realmente no tenía unas probabilidades tan altas.

Por otro lado, algunos juegos también usan esta técnica con otro enfoque. Se trata de juegos que quieren que los resultados aleatorios sean mucho más polarizados. Es decir, que un rango amplio en la aleatoriedad no tiene tanto peso realmente como aparenta, sino que el resultado se decanta mucho más fácilmente por un resultado positivo o uno negativo dependiendo el lado de la balanza en el que estés. Esto quiere decir que si se tiene menos del 50% de posibilidades en el juego, el resultado real es mucho más propenso a fallar, y si es superior, es mucho más propenso a acertar. Es una manera de simplificar los números para que el cerebro de los jugadores lo entienda mejor y no se extrañe si con un 90% de posibilidades falla o con un 10% acierta. De esta manera, prácticamente siempre un 90% en el juego acertará y un 10% fallará. Una manera de hacer esto es simplemente elevando a 2 el porcentaje. De esta manera se “refuerza” el lado de la balanza donde está y se decanta más hacia ese extremo. Esto también hace que los jugadores se acostumbren a hacer acciones más seguras, ya que perjudica a aquellos que intentan obtener buenos resultados con una posibilidad inferior al 50%. En general, le quita peso a la aleatoriedad y su factor sorpresa, ya que dificulta que haya resultados improbables. Esta técnica es usada por varios juegos de la saga *Fire Emblem*[19] («True Hit», s. f.).

Milagros

Otra técnica empleada en pos de que el jugador tenga una buena sensación al jugar es ofrecerle pequeños “milagros”. Hay juegos en los que se pretende que el jugador no tenga en cuenta una posibilidad que le beneficia, ya sea porque no se le indica o porque la posibilidad es muy poco probable. Así pues, si el jugador recibe los beneficios de esa posibilidad cuando no se la esperaba o incluso cuando se esperaba algo perjudicial para él en su lugar, se sentirá afortunado y agradecido. Esto es lo que ocurre un ataque

[19] Saga de videojuegos *RPG* tácticos japoneses desarrollada por Intelligent Systems Co., Ltd.

enemigo falla en *Into the Breach*[20]. Es algo que beneficia al jugador y le da "otra oportunidad", pero debido a las bajas probabilidades que tiene de suceder y el riesgo que conlleva, no es algo que el jugador deba tener en cuenta en su estrategia (Eggplant, s. f.; The Two Types of Random in Game Design - YouTube, s. f.).

Las propiedades del medio

A la hora de realizar acciones aleatorias, es importante tener en consideración el medio y las propiedades de esa aleatoriedad usada. En el caso de los juegos de mesa, muchos usan como medio objetos físicos como dados, cartas o una moneda. En el caso de juegos digitales se usan algoritmos matemáticos que pueden emplearse como el juego requiera, aunque hay veces en las que se intenta replicar el comportamiento de unos dados o cartas.

En el caso de usar acciones similares a una tirada de dados, cada generación aleatoria es independiente de la anterior, cada tirada no se ve influenciada por las anteriores (*probabilidades independientes*). En el caso de usar acciones similares a robar una carta de un mazo, hay que tener en cuenta que en este caso se usa una aleatoriedad con *probabilidades dependientes* según las cartas que restan en el mazo o, dicho de otra forma, las que ya han sido robadas con anterioridad o eliminadas de este. Es decir, si el mazo tiene 2 cartas de cada tipo y el jugador ya ha sacado una de un tipo específico, si vuelve a sacar otra, es menos probable que saque la otra carta restante de ese tipo en específico; mientras que es más probable que saque una carta de un tipo diferente, simplemente por el hecho de que queda más cantidad (The Two Types of Random in Game Design - YouTube, s. f.).

Respecto a esto, hay videojuegos que conservan el funcionamiento y estética de las cartas, pero se deshacen de esta característica e implementan un mazo "infinito" en el que no hay una cantidad limitada de cartas que determine sus probabilidades. Por ejemplo, los mazos de cartas de *Stacklands*[21].

La ilusión del control

Un efecto presente en la aleatoriedad, desde el punto de vista de la psicología, es que se puede aplicar la ilusión de control en esta. Es decir, en lugar de que un número o

[20] Videojuego de estrategia por turnos desarrollado por Subset Games y publicado en 2018.
[21] Videojuego de simulación y gestión de recursos basado en cartas. Fue desarrollado por Sokpop Collective y publicado en 2022.

elemento le sea otorgado a una persona de manera aleatoria, se le puede dar a esa persona la oportunidad de que escoja ese número o elemento por su propia cuenta. A efectos prácticos sigue siendo aleatorio, ya que la persona no tiene conocimiento sobre qué opción es mejor escoger. Sin embargo, esta puede escoger la opción basándose en sus creencias, lo que le puede dar más satisfacción o entretenimiento, además de una falsa ilusión de más control sobre el resultado. Un ejemplo de esto es la lotería, ya que las personas tienen una mejor sensación si escogen por ellas mismas un número, a diferencia de si se les entrega uno aleatorio (Goodman & Irwin, 2006).

El significado sistémico

Un concepto que señala Keith Burgun es el del significado sistémico de los elementos de un juego, entre los que se incluyen aquellos que usan aleatoriedad. El significado sistémico es aquel significado que el jugador le otorga a los elementos del juego a medida que este transcurre. Mide la implicación del jugador con ese elemento, si lo ve con algo con sentido y relacionado con él o, por el contrario, como algo externo o ajeno.

Por ejemplo, si el juego le indica al jugador que un ataque puede fallar o acertar, eso tiene muy poco significado sistémico para el jugador, pero si esa acción está más detallada, justificada y relacionada con él —como que el ataque tiene un 76% de probabilidades de acertar debido al arma que tiene equipada, el personaje y su nivel— tendrá un mayor significado sistémico.

Otro ejemplo sería el caso en donde un jugador ha planteado toda una estrategia en torno a obtener “X” número en un evento aleatorio; ese resultado tiene un significado sistémico para él, de la misma forma en la que se puede generar significado sistémico con “X” número si en dos eventos seguidos le ha tocado como resultado.

Aunque este significado puede formarse por elementos preestablecidos del juego como la narrativa de este, se forma sobre todo después de que el jugador se relacione e interactúe con dichos elementos durante el juego.

(«Three Types of Bad Randomness, and One Good One», 2018)

Determinismo

Un concepto interesante a la hora de diseñar un juego es que debe ser determinista, en el sentido de que, entendiendo el juego como un conjunto de estados de juego a lo largo

del tiempo, las acciones anteriores —de anteriores estados de juego— deben determinar cuál es el estado de juego actual y, junto a las acciones que ocurran en este, deben determinar el futuro de la partida —estados de juego futuros—. Mostrándolo con un ejemplo, no tendría sentido que en una partida de ajedrez, cada "X" turnos las piezas se recolocaran en posiciones distintas a las que los jugadores las han movido, se inutilizarían las acciones de los jugadores.

Cuando se aplican eventos aleatorios en un juego, por su propia naturaleza, estos rompen con el determinismo, con la causa y consecuencia, ya que lo que sucede se genera de forma aleatoria y no está relacionado con lo anterior. Esto hace que, por ejemplo, un acción que lleva preparando un jugador durante mucho tiempo pueda fallar y eso corte con todo lo que se ha estado desarrollando, no solamente en el estado de juego actual sino en los anteriores. De igual manera, esta aleatoriedad no solo modifica el estado de juego actual sino los futuros; claro está, hasta que suceda de nuevo un evento aleatorio que corte con lo anterior.

"Your game is now no longer "A, therefore B, therefore C". Instead, it is now "A, then B, then C"."

(«Randomness and Game Design», 2014)

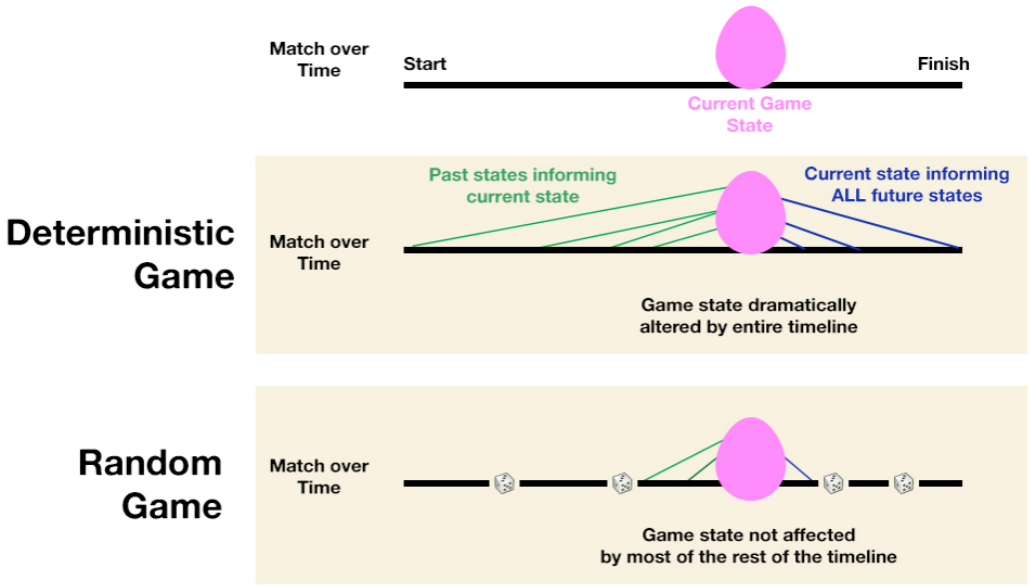

Figura 2. Esquema del determinismo en una partida

En una partida puramente determinista (primer rectángulo), el estado actual del juego es afectado por estados pasados y afectará estados futuros. Cuando se aplica la aleatoriedad, sin embargo, esas conexiones entre estados se cortan (segundo rectángulo).

(«Randomness and Game Design», 2014)

Además, es importante tener en cuenta que el determinismo es precisamente lo que le indica al jugador si está jugando bien o mal, si sus acciones tienen consecuencias positivas o negativas.

(«Randomness and Game Design», 2014)

2.1.10. El determinismo y rigidez del ajedrez

El ajedrez es el claro ejemplo de un juego puramente determinista, ya que todo el estado de la partida se construye en base a las acciones que han realizado los jugadores en turnos anteriores. No hay ningún factor externo a eso (como la aleatoriedad) que modifique el juego. Precisamente por esto, es considerado un juego altamente competitivo en el que dos contrincantes pueden comparar sus habilidades de una forma muy directa e igualitaria.

Por otro lado, debido a esto, también es comprensible que, así como lo hacía Keith Burgun, muchas personas lo vean más como una competición de "ver más lejos" en vez de un juego que permita a los jugadores ser creativos en sus acciones; lo que hace que muchas personas no estén interesadas («Uncapped Look-Ahead and the Information Horizon», 2014). Hay varias formas en las que se podría abordar el problema para intentar hacer el ajedrez —o más bien una variante de este— más divertido para esas personas, aun manteniendo gran parte de su competitividad o *Agon*.

Una de estas consistiría en aplicar algo de aleatoriedad en este, claramente una bien controlada. Un ejemplo de ello son las variantes *shuffle chess* o *chess960* ,(«Fischer Random Chess», 2024) que consisten en empezar la partida de ajedrez con las piezas (excepto los peones) en posiciones aleatorias (aunque en la misma línea del tablero habitual). Para mantener la igualdad, estas posiciones son iguales para ambos oponentes. Concretamente, en el caso de *chess960* se trata de una aleatoriedad más controlada, más uniforme, ya que mantiene normas que se deben cumplir dentro de esa aleatoriedad, como que los dos alfiles de cada jugador estén cada uno en una casilla de un color distinto, como en el ajedrez habitual.

Sin embargo, estas propuestas, pese a ser interesantes, no acaban de cambiar el sistema de juego en sí. Cuando la partida empieza, los jugadores mantendrán el mismo enfoque de una partida común, solo que aplicado a un caso diferente. Pero las reglas

de juego son las mismas. Pasaría algo parecido si se aplicara la aleatoriedad cada vez que una pieza intenta matar a otra, pudiendo acertar o fallar dependiendo de unos porcentajes. Esto sería lo que el autor del artículo *Juegos con aleatoriedad: ¿Es mejor que nos limiten para fomentar la rejugabilidad?* (de 3DJuegos, 2021) denomina como integración vertical del RNG; es decir, aplicar una aleatoriedad superficial a un juego que de por sí tiene un sistema cerrado y sólido, con tal de alterar algunos aspectos de este. Aunque esto podría impulsar una forma de jugar un poco diferente, por ejemplo con jugadas más seguras, esto realmente no logra una mejoría consistente con el sistema de juego base.

Así pues, una forma más eficaz de aplicar la aleatoriedad podría consistir en implementarla en la base del juego, en un nivel mucho más "profundo". Sin embargo, también es importante mantener algo muy reforzado por las bases del ajedrez como lo es la estrategia. Una manera de lograr esto sin que ambos conceptos se opongan directamente es a través de un sistema de *input randomness* que le permita al jugador crear estrategias creativas en base a la aleatoriedad que se le presenta. Un sistema que representa muy bien esto es el de las cartas.

2.1.11. El equilibrio de las cartas

Por otra parte, el correcto funcionamiento de un sistema de cartas depende en gran medida de que se mantenga un equilibrio entre estas. Es un equilibrio frágil que puede truncarse fácilmente si no se tienen en cuenta algunos aspectos generales de los sistemas de cartas y cómo estos se aplican específicamente a un juego determinado.

Una curva de poder (*power curve*) en juegos de cartas es un gráfico que muestra cómo de poderosas son las cartas consideradas promedio en función de su coste. Así pues, si una carta está por encima o por debajo de esta línea, se consideran superiores al promedio o inferiores, respectivamente. Sin embargo, no hay que confundir el poder de una carta con su valor.

Mientras que el poder representa cómo de fuerte o útil es una carta y, por ende, cuánta ventaja te puede llegar a dar en la partida respecto a tu contrincante, el valor es la diferencia entre este poder y el coste de la carta. Por ejemplo, por norma general una carta de coste 6 siempre será más poderosa que una carta de coste 3. Sin embargo, la carta de coste 6 puede ser mucho menos valiosa que la de coste 3 si se tiene en cuenta la relación entre poder y coste.

En muchas ocasiones la curva de poder de las cartas (de menor coste a mayor) no debe ser una línea recta sino una curva o exponencial, ya que las cartas de mayor coste deben tener un valor mayor proporcionalmente que las de menor coste. De esta forma, una carta de coste 7 debe tener un valor superior a una carta de 5 y de 2 juntas. Esto se debe principalmente a 2 motivos:

— Se debe recompensar el riesgo que suponen las cartas de alto costo, ya que es posible que la partida termine (o se decante por uno de los jugadores) antes de que el jugador sea capaz de usarlas.

— Se debe recompensar la inflexibilidad que suponen las cartas de alto costo, ya que las de bajo costo pueden usarse en una mayor cantidad de turnos, desde los más iniciales hasta los últimos. Además, en caso de usarse en los turnos más adelantados, pueden hacerlo como una combinación junto a otras cartas. Las cartas de alto costo, por el contrario, no ofrecen esa flexibilidad.

Así pues, a la hora de balancear el juego, para indicar cómo de buena es una carta no debe mirarse la distancia real (por encima o por debajo) a la que se encuentra respecto la curva de poder, sino el ratio del poder que tiene respecto el poder de la curva. De esta manera, en la siguiente imagen se muestran las cartas 1 (de coste 2) y 2 (de coste 7). Ambas están la misma distancia por encima de la curva de poder, pero el ratio es mayor en la carta 1, por lo que es una “mejor” carta que la carta 2.

Figura 3. Cartas de Hearthstone en una curva de poder

Muestra una comparativa entre el valor de varias cartas de Hearthstone[22] posicionadas en una curva de poder. Imagen extraída de la conferencia de Dylan Mao en la GDC 2018. Está editada para marcar en rojo la carta 1 y la carta 2.

(Board Game Design Day: Balancing Mechanics for Your Card Game's Unique Power Curve - YouTube, s. f.)

También hay que tener en cuenta el sistema de recursos y costes del juego a la hora de hacer la curva de poder, entendiendo los recursos como magia o energía que necesita el jugador para usar sus cartas de diferentes costes. Juegos como *Hearthstone* en los que el jugador va ganando recursos cada vez mejores cada turno de manera automática le darán menos importancia a la pronunciación de la curva. Le dará más poder a las cartas de bajo coste y menos a las de alto, para que los jugadores no esperen simplemente a poder usar cartas de alto coste. En cambio, juegos como *Magic: The Gathering*[23] en los que el jugador debe implicarse y esforzarse para conseguir mejores recursos, tendrán una curva de poder más pronunciada donde las cartas de bajo coste tendrán un menor poder y las de coste alto tendrán mucho más, ya que de esta manera se incita más a los jugadores a ganar los recursos necesarios para jugarlas (y se les recompensa por hacerlo).

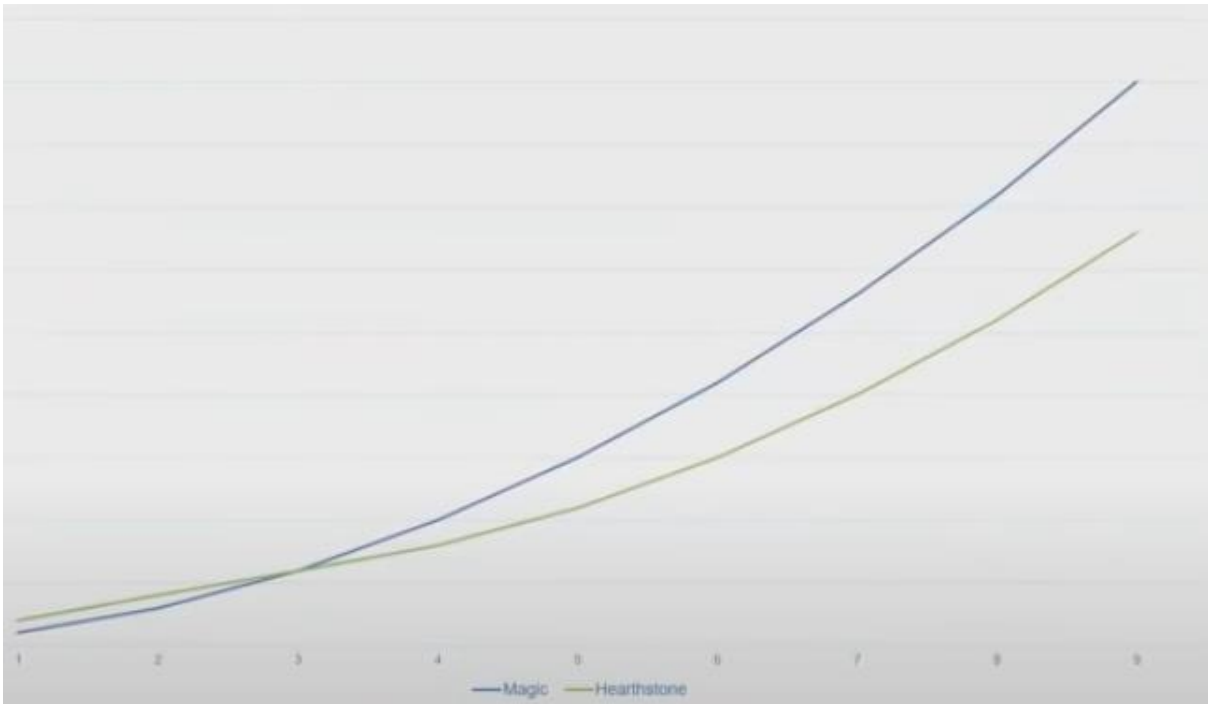

Figura 4. Curvas de poder de Hearthstone y Magic: The Gathering

[22] Videojuego *online* de cartas desarrollado por Blizzard Entertainment y publicado en 2014.
[23] Juego de cartas coleccionables (CCG) creado por Richard Garfield y publicado en 1993.

Representa aproximadamente como se verían las curvas de poder de Hearthstone (verde) y Magic: The Gathering (azul). Imagen extraída de la conferencia de Dylan Mao en la GDC 2018.

(*Board Game Design Day: Balancing Mechanics for Your Card Game's Unique Power Curve - YouTube*, s. f.)

Por otro lado, juegos como "Pokémon TCG" tienen un sistema de recursos y costes muy diferentes donde no necesariamente se deben tener recursos altos para usar cartas más poderosas o viceversa, ya que no existen tales recursos. El coste de jugar cartas más o menos poderosas depende de otros factores del juego como que se den ciertas condiciones en el turno o baraja de cartas o incluso es un coste que se paga después de usar la carta y no antes.

Por último, juegos como *Clash Royale*[24] tienen otro sistema distinto de recursos y costes. En este sí que hay unos costes en las cartas que dependen de los recursos del jugador, pero la utilidad de las cartas depende mucho más de la situación en la que se jueguen (por ejemplo para contrarrestar cartas enemigas o engañar al rival) que del coste que estas tienen de por sí. Su valor puede ser extremadamente alto o bajo dependiendo de la situación en la que se juegue, por lo que resulta más difícil definir su valor predeterminado.

(*Board Game Design Day: Balancing Mechanics for Your Card Game's Unique Power Curve - YouTube*, s. f.)

2.1.12. Ajedrez y cartas

Con todo esto dicho, es posible hacerse una idea de cómo debería comportarse un sistema de cartas que busque un buen equilibrio. Sin embargo, si se pretende incorporar en un sistema de juego con una estrategia como la del ajedrez, se debe trabajar especialmente en la cohesión de ambos sistemas y cómo se entrelazan estos para conseguir lo que se busca.

[24] Videojuego *online* de estrategia en tiempo real para móviles, desarrollado por Supercell y publicado en 2016.

2.2. Estudio de mercado

2.2.1. Productos similares

Este proyecto intenta alcanzar un buen equilibrio entre la estrategia del ajedrez y la aleatoriedad de las cartas. Actualmente, ya se han lanzado al mercado algunos productos que, de una manera más o menos directa, se involucran en este tema. Algunos de los más destacados pueden ser el propio ajedrez al que se hace referencia en el estudio, videojuegos de cartas populares como *Hearthstone* o *Slay the Spire*, o juegos que intentan combinar el ajedrez y la aleatoriedad de una manera más directa, como *Dice Chess*, *No Stress Chess*, *Uno Chess*, *Not Chess* o *Knightmare Chess*. A continuación se muestra un análisis de los productos mencionados.

Ajedrez

Figura 5. Ajedrez

(«Chessboard», 2024)

Descripción:

El ajedrez es un juego de mesa clásico de estrategia. Destaca por su competitividad y su sistema para enfrentar las habilidades de los jugadores de una manera justa y transparente.

Público:

Cualquier edad, recomendado a partir de 5 años.

Público general, sobre todo aquellas personas más competitivas y/o a la que les gusten los juegos intelectuales.

Lanzamiento:

No hay certezas sobre su origen.

La versión web más popular del juego es https://www.chess.com/ (Chess.Com - Play Chess Online - Free Games, s. f.).

Jugadores:

Se enfrentan 2 jugadores.

Mecánicas:

La única acción que estos pueden llevar a cabo una vez por turno es mover una pieza, en algunos casos haciendo que esta capture una del rival.

Objetivo:

El objetivo es “amenazar” al rey enemigo (jaque) y dejar al rival sin posibilidad de evitar esa amenaza (jaque mate).

Aleatoriedad y balance:

No hay aleatoriedad ni limitaciones impuestas en la información que tienen los jugadores. Se puede considerar que el jugador con piezas blancas tiene una ligera ventaja en la partida por empezar primero.

(«Chessboard», 2024)

Hearthstone

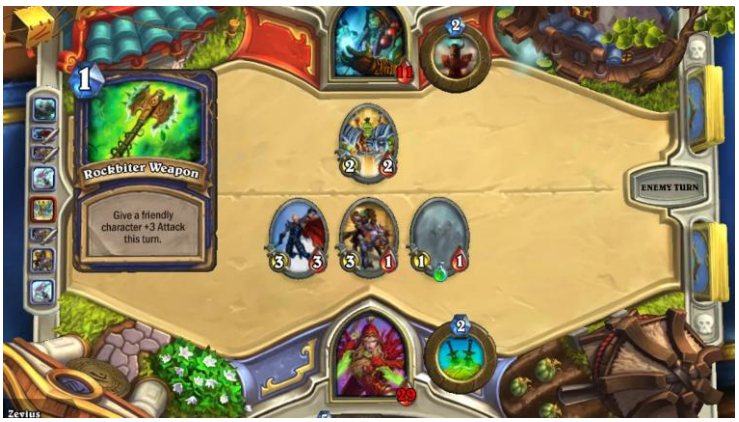

Figura 6. Hearthstone

(Parker, s. f.)

Descripción:

Hearthstone es uno de los videojuegos de cartas más populares. Destaca por ser un juego relativamente sencillo y accesible para nuevos jugadores y por ofrecer situaciones muy variadas debido a la aleatoria en sus partidas.

Público:

Cualquier edad, recomendado a partir de 7 años.
Público general, desde jugadores de videojuegos casuales hasta competitivos, a los que les gusten los juegos intelectuales.

Lanzamiento:

Fue desarrollado por Blizzard Entertainment y lanzado en 2014. Está disponible en Windows, macOS, iOS y Android.

Jugadores:

Se enfrentan 2 jugadores.

Mecánicas:

Los jugadores, en su turno, pueden, principalmente, jugar diferentes cartas de su mano, atacar con sus “piezas” llamadas esbirros o usar una habilidad propia de su personaje (poder de héroe). Las cartas pueden ser de dos tipos, hechizos (activan algún efecto) o esbirros (invocan esbirros en el “tablero”).

Objetivo:

El objetivo es bajar los puntos de vida del enemigo a 0 (o menos).

(«*Hearthstone*», 2024)

Aleatoriedad y balance:

La aleatoriedad está presente en muchos aspectos del juego y se presenta con tipos y características variadas. Los mayores exponentes de esta son el *input randomness* de robar cartas y el *output randomness* de algunos efectos de esbirros o cartas cuando se usan.

Slay the Spire

Figura 7. Slay the Spire

(Donlan, 2019)

Descripción:

Slay the Spire es considerado un videojuego pionero en combinar un sistema de cartas con el género *roguelike*. Además, obtuvo varias nominaciones y elogios por su sistema de juego estratégico bien diseñado.

Público:

Cualquier edad, recomendado a partir de 7 años.
Público general, aunque se puede considerar un juego más de nicho y no tan casual.
Para jugadores que busquen experiencias nuevas y retos, sobre todo a nivel intelectual.

Lanzamiento:

Fue desarrollado por Mega Crit y lanzado en Windows, macOS, Linux, PlayStation 4, Nintendo Switch y Xbox One en 2019, en iOS en 2020 y en Android en 2021.

Jugadores:

Es un juego de un solo jugador en el que este se enfrenta a la consola y a varios elementos aleatorios.

Mecánicas:

El jugador, en su turno, puede, principalmente, jugar las cartas de su mano o usar algunos objetos. Las cartas esencialmente sirven para protegerte de ataques enemigos o infligir daño a estos, además de algunos otros efectos. Si en vez de centrarse en el

combate se considera toda la *run* como la partida, el jugador también debe tomar diferentes decisiones que involucran sus cartas jugables u objetos con distintos efectos en los combates.

Objetivo:

El objetivo de cada combate es bajar los puntos de vida de los enemigos a 0 (o menos). Su objetivo de la run es llegar hasta la sala final del mapa y derrotar al boss.

(«*Slay the Spire*», 2024)

Aleatoriedad y balance:

La aleatoriedad se presenta en el juego mayormente como *input randomness* al robar cartas en los combates y al hacerte elegir entre opciones fuera de estos. Aunque también puede haber *output randomness* en algunos elementos, esta es mínima en comparación, ya que precisamente el juego se centra mucho en que el jugador construya estrategias en base al *input randomness*, y potencia esto haciendo posible varias combinaciones de cartas que se complementan.

Dice Chess

Figura 8. Dice Chess

(thebeanone, 2023)

Descripción:

Dice Chess es una variante del ajedrez que incluye tiradas de dados en el juego. Aunque hay distintas variaciones de esta, una popular consiste en que los jugadores tiran dos

dados en cada turno. Dependiendo qué haya salido en estos, el jugador debe escoger uno de ellos y mover un tipo de pieza que está asociada con ese resultado (en caso de dados 1-6 comunes, 1 puede significar peón, 2 caballo, etc., aunque también hay dados que tienen los iconos de las piezas directamente). En caso de que ambos dados den el mismo resultado, se le da libertad al jugador de mover la pieza que quiera.

Público:

Cualquier edad, recomendado a partir de 5 años.
Público general, aunque se puede considerar un juego más de nicho y no tan casual.
Para personas que busquen nuevas formas de jugar al ajedrez.

Lanzamiento:

No hay certezas sobre su origen.

Jugadores:

Se enfrentan 2 jugadores.

Mecánicas:

El jugador puede mover las piezas de la misma manera que lo hace en el ajedrez normal, solo que está limitado sobre cuándo hacerlo. Además, se añade la acción de tirar los dados y la de escoger la pieza según estos.

Objetivo:

La meta sigue siendo amenazar al rey enemigo al igual que en el ajedrez normal. Por otra parte, no hay jaque ni jaque mate como tal, sino que se le debe capturar como cualquier otra pieza. Esto se consigue si el enemigo no es capaz de esquivar esa amenaza, lo que puede depender en gran parte de las tiradas de dados.

(«Dice Chess», 2023; *How to play Dice Chess - YouTube*, s. f.)

Aleatoriedad y balance:

Se aplica la aleatoriedad en el ajedrez en forma de un *input randomness* aparentemente puro. A pesar de que se puede considerar una buena decisión el hecho de ofrecer al jugador esta libertad de escoger entre dos resultados de los dados, las opciones que tiene siguen estando muy limitadas. No solo se ven limitadas las piezas que el jugador puede mover en este turno, sino también las que podrá mover en los turnos futuros.

Esto le imposibilita mucho el hecho de crear y poder seguir una estrategia. Una vez se tiene construida una estrategia, las tiradas futuras de dados se convierten, en cierta parte, en *output randomness*, ya que simplemente determinan si se tiene éxito en proseguir con la estrategia o no, a partir de resultados aleatorios. Cada tirada de dados puede romper fácilmente con el determinismo del juego y sus estados futuros, lo que además representa un *flow* de información con una frecuencia muy alta. Además, en caso de que el rey esté amenazado, esta aleatoriedad tan variable (no controlada) influye enormemente en el resultado final de la partida (tiene un delta muy amplio), lo que puede resultar injusto.

No Stress Chess

Figura 9. No Stress Chess

(«No Stress Chess Game Just $12.40 on Amazon (Reg. $20) | Over 5,900 5-Star Ratings», 2023)

Descripción:

No Stress Chess es un juego de mesa que intenta modificar algunos aspectos de cómo se desarrolla la partida a través de cartas, las cuales indican un tipo de pieza y sus posibles movimientos. Tiene dos variantes, una más enfocada a aprender a jugar y otra en la que se le da más libertad de decisión a los jugadores. En la primera, los jugadores deben empezar con la apertura más clásica del ajedrez. A partir de ahí, una vez por turno, deben mostrar una carta de las del mazo y mover una pieza del tipo indique. En la segunda variante, los jugadores disponen cada uno de una mano de cartas entre las que pueden escoger una cada turno.

Público:

Cualquier edad, recomendado a partir de 3 años.

Público general, aunque está más enfocado a un público muy joven que busca aprender a jugar al ajedrez, sobre todo usando su primera variante de juego.
Para personas que busquen nuevas formas de jugar al ajedrez.

Lanzamiento:

Fue desarrollado por Todd Kurtzer y John Zaruba y lanzado en 2004.

Jugadores:

Se enfrentan 2 jugadores.

Mecánicas:

El jugador puede mover las piezas de la misma manera que lo hace en el ajedrez normal, solo que está limitado sobre cuándo hacerlo, así como también ocurren en el *Dice Chess*. Además, en la segunda variante, se añade la acción de escoger la carta que se quiera usar en ese turno, de entre las de la mano del jugador.

Objetivo:

La meta sigue siendo amenazar al rey enemigo al igual que en el ajedrez normal. Por otra parte, no hay jaque ni jaque mate como tal, sino que se le debe capturar como cualquier otra pieza. Esto se consigue si el enemigo no es capaz de esquivar esa amenaza, lo que puede depender en gran parte de las cartas.

(*No Stress Chess*, s. f.; *No Stress Chess : Amazon.co.uk: Toys & Games*, s. f.; Triple S Games, 2021)

Aleatoriedad y balance:

La aleatoriedad se implementa de una manera muy parecida a como lo hace *Dice Chess* y tiene unas consecuencias prácticamente iguales. Sin embargo, en la primera variante las limitaciones que tiene el jugador son incluso mayores a las de *Dice Chess*, ya que este no tiene opción de escoger entre dos resultados (de dos dados), sino que el tipo de pieza que debe usar ya le viene dado por la carta directamente. Así pues, se puede interpretar que constituye un peor diseño, ya que deja prácticamente sin libertad al jugador. Por otra parte, también se debe tener en cuenta que se trata de una variante enfocada al aprendizaje del juego.

Por el contrario, la segunda variante, aún y mantener en gran parte el funcionamiento de *Dice Chess*, hace algunas cosas diferentes, las cuales se pueden considerar como mejores en algunos aspectos. Lo primero es que el número de opciones del jugador es mayor, ya que este dispone de 4 cartas en su mano, a pesar de que puede que algunas de estas se repitan. Lo segundo y más importante a nivel de diseño, es el propio formato de cartas, que permite que los jugadores mantengan sus cartas en su mano durante varios turnos. Así, estos pueden plantear como usarlas en los turnos futuros y crear un plan o estrategia para decidir el orden en el que las usan, cual deberían descartar o guardar para más adelante, etc. Es un juego mucho más determinista que *Dice Chess*.

Uno Chess

Figura 10. Uno Chess

(Triple S Games, 2023a)

Descripción:

Uno Chess es una variante del ajedrez en la que se juega al popular juego de cartas Uno. En cada turno, los jugadores deben jugar una carta siguiendo las reglas clásicas del juego (aunque cuando se desprenden de ella, deben robar otra que la sustituya). Cuando lo hagan, dependiendo el número de esta, los jugadores podrán mover una pieza de ajedrez que se encuentre en una casilla de la fila o columna correspondiente a ese número. Además, las cartas de tipo “wild” permiten que el jugador mueva la pieza que quiera y las “reverse” que el oponente deshaga su último movimiento. Las de tipo “+2” hacen que el jugador pueda cambiar 2 de sus cartas por otras del mazo.

Público:

Cualquier edad, recomendado a partir de 5 años.

Público general, aunque se puede considerar un juego más de nicho y no tan casual.

Para personas que busquen nuevas formas de jugar al ajedrez.

Lanzamiento:

No hay certezas sobre su origen.

Jugadores:

Se enfrentan 2 jugadores.

Mecánicas:

El jugador puede mover las piezas de la misma manera que lo hace en el ajedrez normal, solo que está limitado sobre cuándo hacerlo, así como también ocurren en el *Dice Chess* o *No Stress Chess*. Además, se añade la acción de escoger la carta que se quiera usar en ese turno, de entre las de la mano del jugador.

Objetivo:

La meta sigue siendo amenazar al rey enemigo al igual que en el ajedrez normal. Por otra parte, no hay jaque ni jaque mate como tal, sino que se le debe capturar como cualquier otra pieza. Esto se consigue si el enemigo no es capaz de esquivar esa amenaza, lo que puede depender en gran parte de las cartas.

(Triple S Games, 2023a)

Aleatoriedad y balance:

La aleatoriedad se implementa de una manera muy parecida a como lo hace la segunda variante de *No Stress Chess*, ya que se permite que los jugadores creen estrategias a futuro con las cartas de su mano. Algo que puede constituir un mejor diseño es el hecho de que las cartas no definen tipos de piezas, sino casillas en el tablero. Esto puede permitir crear unas estrategias con más profundidad, ya que permite enlazar de una manera más directa varias cartas (por ejemplo, si el jugador mueve una pieza a una casilla que sabe que podrá usar en el siguiente turno gracias a otra de sus cartas). Además le da otro enfoque a las estrategias, como mantener las piezas dispersas para aumentar las probabilidades de poderlas usar con las cartas. Como punto negativo, puede dar la impresión de que ambos juegos se combinan de una manera un poco forzada, además de que tienen naturalezas muy distintas y Uno se rige bastante por la

aleatoriedad, por lo que es probable que en muchos turnos los jugadores no tengan cartas que puedan usar y su turno se desaproveche (en ese caso usan una carta aunque no tenga relación con la actual, pero no se les permite mover ninguna pieza).

Not Chess

Figura 11. Not Chess

(*Thanks for letting me check this out @NOT CHESS ! #chess #boardgame #k...* | *TikTok*, s. f.)

Descripción:

Not Chess fue una propuesta de juego de mesa que aplicaba un sistema de cartas a un tablero y piezas de ajedrez. Sin embargo, las piezas no disponían de sus movimientos ni características típicas, sino que una actuaba como reina (su movimiento era igual al de una reina de ajedrez) y las demás como “no reinas” (su movimiento consistía en moverse a una casilla adyacente y no podían capturar piezas enemigas). La propuesta del juego era darle mucha más importancia a las cartas y sus efectos.

Público:

Aunque no salió a la venta, se puede aproximar que era apto para cualquier edad, recomendado a partir de 5 años.
Público general, aunque se puede considerar un juego más de nicho y no tan casual.
Para personas que busquen nuevas formas de jugar al ajedrez, no tan intelectuales sino más “divertidas” y “alocadas”.

Lanzamiento:

Fue desarrollado por Goodlark Games y lanzado como campaña de crowdfunding en Kickstarter en 2022. Sin embargo, no obtuvo la recaudación suficiente y no se lanzó al mercado finalmente.

Jugadores:

Se enfrentan de 2 a 6 jugadores, en dos equipos.

Mecánicas:

El jugador, durante su turno, puede elegir si mover una pieza con su correspondiente rango de movimiento, usar una carta, robar una del mazo o reemplazar algunas de las de la mano (hasta 3).

Objetivo:

La meta es eliminar todas las piezas enemigas del tablero.

(Jed Herne, 2022; Karar2k's Why Play This, 2022; *NOT CHESS*, 2023)

Aleatoriedad y balance:

Puesto que el juego no salió al mercado es difícil analizar su funcionamiento. Sin embargo, su sistema de juego se diferencia más de los casos vistos anteriormente. Se aleja más de la idea del ajedrez y focaliza su *gameplay* especialmente en los efectos de las cartas. Además, su objetivo también es distinto; se trata de eliminar todas las piezas enemigas en vez de una sola como lo es el rey. Esto puede ser considerado una buena decisión si se tiene en cuenta que los efectos de las cartas se enfocan precisamente a eso y/o pueden implicar situaciones no deseadas en caso de que el objetivo fuese capturar una pieza concreta. Por ejemplo, hay cartas que permiten eliminar piezas al momento, sin que el enemigo pueda hacer nada al respecto o lo pueda prever, lo que sería injusto si esa pieza fuese muy importante, como el rey, y eso decidiera la partida.

Por otro lado, el juego se muestra simple; tanto por sus normas como por la profundidad que pueda tener una partida. Que las normas sean simples no es necesariamente malo. Sin embargo, sí que puede resultar un problema lo segundo, ya que no parece que se puedan implementar estrategias profundas con el tipo de cartas que se presentan. Estas cartas se basan principalmente en mover y eliminar fichas, pero no parece que se pueda trabajar mucho con ellas para formar combinaciones interesantes entre ellas que permitan realizar jugadas con profundidad. Sin esta profundidad en las jugadas (y en los

elementos de juego en general, como lo son los dos únicos tipos de fichas), se puede hacer un juego entretenido por un corto tiempo, pero no uno que mantenga a los jugadores interesados tras varias partidas.

Knightmare Chess

Figura 12. Knightmare Chess

(Triple S Games, 2023b)

Descripción:

Knightmare Chess es una variante del ajedrez que aplica cartas a la partida, las cuales tienen efectos variados. Estos efectos pueden ser instantáneos (suceden en el mismo turno) o continuos (se mantienen durante la partida o hasta que sucede alguna circunstancia que lo detiene). Cada carta indica en qué momento se puede jugar, pero siempre será durante el turno del jugador que la usa y no podrá usar más de una. Algunas cartas impiden que el jugador mueva una pieza. Algunos efectos destacados consisten en alterar las características de algunas piezas o reincorporar en el tablero piezas que habían sido capturadas anteriormente, pero hay una gran variedad de efectos. Sin embargo, los efectos de las cartas no pueden ser utilizados para capturar o hacer jaque mate al rey enemigo directamente (si no se trata de un efecto continuo), sino que esto se debe conseguir a través de una jugada de una pieza.

Público:

A partir de 3 años, recomendado a partir de 5 años.
Público general, aunque se puede considerar un juego más de nicho y no tan casual.

Para personas que busquen nuevas formas de jugar al ajedrez o les llamen la atención las cartas. Además, las personas no deben intimidarse por la aparente complejidad del juego y todas sus cartas, así como su estética más “seria” y arte “tenebroso”.

Lanzamiento:

Knightmare Chess es la versión americana del juego francés *Tempête sur l'échiquier,* aunque el arte y las cartas de ambos juegos difieren. Hay un total de 160 cartas publicadas por Steve Jackson Games, 80 pertenecientes a un primer set (*Knightmare Chess 1*) en 1996 y las otras 80 pertenecientes a un segundo (*Knightmare Chess 2*) en 1997.

Jugadores:

Se enfrentan 2 jugadores.

Mecánicas:

El jugador, durante su turno, puede mover una pieza de ajedrez de forma común (o no, dependiendo si se le aplica el efecto de alguna carta) y/o usar una carta siempre que su descripción lo permita. También puede reemplazar una carta de su mano por una del mazo.

Objetivo:

La meta es hacer jaque mate al rey enemigo.

(«*Knightmare Chess*», 2023; Triple S Games, 2023b)

Aleatoriedad y balance:

La aleatoriedad se aplica de manera correcta en el juego. Permite a los jugadores crear estrategias como lo harían en una partida de ajedrez normal, pero le añade la diversidad de las cartas y toda la profundidad que conllevan sus efectos. Además, evita situaciones importantes injustas que uno de los jugadores no puede prever, como lo sería un jaque mate directo por el efecto de una carta que se usa en ese mismo turno. Por otra parte, su sistema de cartas se basa en que ambos jugadores crean un mazo para la partida (por lo general, distinto uno del otro), lo cual se aleja de un juego más simétrico en el que los dos jugadores juegan con las mismas cartas, o por lo menos tienen esa posibilidad.

2.2.2. Conclusiones

Aunque los juegos expuestos son, en su mayoría, aptos para un rango amplio de público, muchos quedan como juegos de nicho por el poco alcance que llegan a conseguir, quedando reducido su grupo de jugadores, en muchos casos, a aquellos que son experimentados en el ajedrez clásico y buscan otras variantes y formas nuevas de jugar a este. Sin embargo, el medio de los videojuegos permite llegar a un público más extendido con más facilidad que con un juego de mesa o pack de cartas, como en el caso de *Hearthstone* o *Slay the Spire*, que son juegos mucho más conocidos entre el público general que *Uno Chess*, *Knightmare Chess* o cualquier otro juego de mesa mencionado.

Además, entre *Hearthstone* y *Slay the Spire* el primero es mucho más conocido que el segundo. Esto se puede deber a varios factores: es gratuito (más accesible), es *online* (mayor relación entre jugadores), es desarrollado por una gran compañía (mayor reputación y presupuesto) y es aparentemente más simple (accesible para un jugador nuevo y/o casual).

También hay que tener en cuenta que la mayoría de los juegos no son recientes, siendo *Knightmare Chess* el que más se acerca a la idea de este estudio pero habiendo sido publicado en 1996, y siendo *Not Chess* el más reciente pero no habiéndose publicado finalmente.

En cuanto a las mecánicas y sistemas de juego que ofrece cada uno de los productos, se pueden sacar varias conclusiones, siendo estas algunas destacadas:

— Una sencillez como la de *Hearthstone* puede ser muy atractiva y accesible para los jugadores, aunque su *output randomness* es algo criticado por algunas personas.

— Un sistema de cartas como el de *Slay the Spire* puede funcionar muy bien. En este, el jugador plantea qué estrategias quiere seguir en base a las cartas que tiene y las que consigue y sus afinidades. Él escoge el rumbo que quiere seguir en la partida.

— Un sistema de dados como el de *Dice Chess* puede acarrear varios problemas a nivel de diseño. Puede limitar demasiado al jugador, romper el determinismo de la partida y actuar como un *output randomness* no deseado y con un delta muy amplio. Por el contrario, un sistema de cartas como el de *No Stress Chess*, siempre que se usen varias

cartas y no una sola (como ocurre en su primera variante), es preferible porque corregiría estas problemáticas.

— Un sistema de cartas como el del *Uno Chess* puede ofrecer más profundidad en las jugadas y estrategia que el de *No Stress Chess*, debido a que las cartas indican posiciones en vez de casillas. Esto se puede extrapolar a cualquier juego, donde se puede ampliar la profundidad de una mecánica si se enfoca desde otro punto de vista.

— El sistema de juego de *Not chess* deposita mucho más peso en las cartas que en el tablero, sin embargo esas cartas no permiten unas estrategias sólidas y complejas por ellas mismas. Por otra parte, el tablero y piezas son extremadamente simples y con poca profundidad. Esto puede hacer que el juego no tenga la profundidad suficiente como para ser jugado varias veces. Sin embargo, el objetivo del juego (eliminar todas las piezas enemigas, en vez de capturar al rey) está bien diseñado, ya que se ajusta a los efectos de las cartas y el funcionamiento de la partida.

— El sistema de juego de *Knightmare Chess* presenta un buen diseño, simple y efectivo, en cuanto a la aplicación de la aleatoriedad. Sin embargo, hacer que los oponentes jueguen con el mismo mazo de cartas podría hacer la partida más justa. Además, gran parte de su complejidad reside simplemente en los efectos individuales de las cartas. Otra manera de enfocarlo sería a través de alguna otra mecánica en el sistema. Por otra parte, existe una norma que refuerza el buen diseño del juego. Se trata de “prohibir” jugadas de cartas que capturen al rey enemigo directamente. Esta es una limitación que evita un delta muy grande e injusto para uno de los jugadores.

Estas conclusiones se tendrán en cuenta durante el desarrollo del proyecto.

3. Gestión del proyecto

La gestión del proyecto puede dividirse en dos aspectos principales. La *planificación* realizada antes de empezar con los procesos de desarrollo y el *seguimiento* de estos mismos procesos en el momento de llevarlos a cabo.

3.1. Planificación

3.1.1. Objetivos

Los principales objetivos que cumplir con tal de crear un prototipo de juego, en orden de desarrollo, son los siguientes:

1. Diseño:

- Diseñar el sistema de juego
- Diseñar las interfaces

2. Programación:

- Programar la conectividad *online*
- Programar las interfaces
- Programar las partidas

3. Pulido:

- Creación del arte
- Añadido de funcionalidades y detalles (chat, personalización, etc.)

4. Análisis del resultado:

- Validación del juego

Los distintos objetivos están separados en varios bloques. El orden en el que se llevarán a cabo es bastante representativo en cuanto a lo que se pretende conseguir con el resultado final. Así pues, se prioriza, respectivamente, un buen diseño de juego, un correcto funcionamiento de este y, por último, una buena estética y funcionalidades que complementen el prototipo y lo hagan verse como algo más completo y acabado. Finalmente, se obtendrán ciertas valoraciones del juego, se analizará su estado y se idearán posibles mejoras.

3.1.2. Recursos

Los recursos necesarios para llevar a cabo este proyecto son relativamente asequibles, pues no se requiere ninguna tecnología o software especialmente complejos para lograr los distintos objetivos planteados. Por otro lado, un mínimo de un *ordenador de gama media-alta* es imprescindible y *dos pantallas* en vez de una mejorarían la comodidad y agilizarían mucho el ritmo de trabajo, así como también lo harían algunos programas de diseño gráfico como *Adobe Illustrator*[25].

3.1.3. Tiempo

En cuanto al tiempo disponible para realizar el desarrollo del proyecto, se trata de aproximadamente 100 días desde la primera entrega de la memoria del trabajo (22 de marzo de 2024) hasta la última (30 de junio de 2024), siendo este el periodo más focalizado a desarrollar el prototipo del juego.

3.1.4. Análisis inicial de costes

Para obtener una aproximación de los costes del proyecto, se tendrán en cuenta los costes únicos y mensuales involucrados en este.

Los costes únicos (*ver Tabla 2*) son aquellos que se realizarán una sola vez en todo el proyecto. Están compuestos por el ordenador donde se realizará todo el trabajo y la pantalla adicional que lo facilitará. Además, si se pretende que el resultado final esté lo más pulido posible, se deberá integrar en la tienda de Steam como un juego completo[26]. Este proceso tiene un coste fijo de 100 dólares que se ha aproximado a 95 euros. Sin embargo, hay que tener en cuenta que, en caso de monetizar el juego y de que este genere 1.000 $, los 100 $ iniciales son reembolsados. El total de costes únicos es de 1.395 €.

Costes únicos:

Ordenador	1.200
Pantalla adicional	100

[25] Software de edición de gráficos vectoriales muy usado por diseñadores gráficos. Fue desarrollado por Adobe Inc. y publicado en 1987.
[26] Esto es debido a las librerías Steamworks que usa el juego con tal de conectar a los jugadores vía *online*, además de otras funcionalidades. Al integrar el juego en el catálogo de Steam, se puede sacar un mayor partido a estas librerías. Para más información, ver el apartado *5.3. Programación de la conectividad online*.

Publicación en Steam	95
Total	1.395

Tabla 2. Costes únicos

Los costes mensuales (*ver Tabla 3*) son aquellos que se repiten cada mes. En estos se ha tenido en cuenta un sueldo promedio de Junior Game Developer en Barcelona (*Sueldo*, s. f.). Además, también se incluye la licencia mensual del programa Adobe Illustrator para permitir un trabajo creativo ágil. El total de costes mensuales es de 2.127 €.

Costes mensuales:

Junior Game Developer	2.100
Licencia Adobe Illustrator	27
Total	2.127

Tabla 3. Costes mensuales

Los costes mensuales totales (*ver Tabla 4*) son aquellos que se obtienen al multiplicar los costes mensuales por el número de meses en el que se trabajará. El periodo de trabajo cubre 4 meses en los que Adobe Illustrator será requerido. Por otra parte, el tiempo de trabajo del Game Developer se cuantifica por días, los cuales no llegan a constituir 4 meses en su plenitud. A pesar de esto, este tiempo se aproxima a los 4 meses si se tiene en cuenta el trabajo previo de conceptualización e investigación. El total de costes mensuales totales es de 8.508 €.

Costes mensuales totales:

Costes mensuales	2.127
Número de meses	4
Total	8.508

Tabla 4. Costes mensuales totales

Finalmente, los costes totales del proyecto (*ver Tabla 5*) se obtienen al sumar los costes mensuales totales y los costes únicos. Así pues, el resultado es de **9.903 €**.

Costes totales:

Costes mensuales totales	8.508
Costes únicos	1.395
Total	9.903

Tabla 5. Costes totales

3.1.5. Diagrama de Gantt

En la siguiente tabla (*ver Tabla 6*) se muestra una información más detallada sobre la duración prevista para las distintas tareas. Además de las tareas de desarrollo, se muestran en rojo las *deadlines* y en naranja y amarillo los periodos de tiempo en el que se tendrán más dificultades y facilidades a la hora desarrollar el proyecto, respectivamente (*Dificultades externas* y *Facilidades externas*). Esto es debido a factores externos en su mayoría académicos, como entregas de trabajos finales en las que se prevé tener una carga de trabajo grande (menos tiempo a invertir) y, por el lado opuesto, las vacaciones posteriores a esto (más tiempo a invertir).

Seguido de esto se encuentran las tareas de desarrollo mencionadas, en donde se le otorga un gran peso a *Programar las partidas del juego*. Esto es debido a que se espera poder contar con un tiempo de programación suficiente como para estar seguros de poder implementar todo lo diseñado correctamente y hacer frente a *bugs* y problemas que surjan durante esta misma etapa. Además, también se considera la posibilidad de llevar a cabo algunos cambios respecto al diseño original después de poder *testear* el funcionamiento del juego en tiempo real, por lo que es conveniente tener un buen margen para ello. Finalmente, otro motivo es el hecho de que los inicios de esta etapa se verán afectados por las ya mencionadas dificultades externas, por lo que se espera un mayor rendimiento en su segunda mitad, donde se verá beneficiada por las facilidades.

ID	Name	Start Date	End Date	Duration	Color
1	Deadlines	Mar 22, 2024	Jun 30, 2024	100 days	
2	Deadline 1	Mar 22, 2024	Mar 22, 2024	0 days	
3	Deadline 2	May 10, 2024	May 10, 2024	0 days	
4	Deadline 3	Jun 30, 2024	Jun 30, 2024	0 days	
5	Dificultades externas	Apr 22, 2024	May 15, 2024	24 days	
6	Facilidades externas	May 15, 2024	Jun 30, 2024	47 days	
7	Desarrollo	Mar 22, 2024	Jun 29, 2024	100 days	
8	Diseñar el sistema de juego	Mar 22, 2024	Mar 31, 2024	10 days	
9	Diseñar la interfaz del juego	Apr 01, 2024	Apr 04, 2024	4 days	
10	Programar la conectividad Online	Apr 05, 2024	Apr 14, 2024	10 days	
11	Programar las interfaces del juego	Apr 15, 2024	Apr 21, 2024	7 days	
12	Programar las partidas del juego	Apr 22, 2024	May 31, 2024	40 days	
13	Creación del arte del juego	Jun 01, 2024	Jun 10, 2024	10 days	
14	Añadido de funcionalidades y detalles	Jun 11, 2024	Jun 20, 2024	10 days	
15	Validación del juego	Jun 21, 2024	Jun 29, 2024	9 days	

Tabla 6. Tareas a realizar

(Free Online Gantt Chart Software, s. f.)

Toda esta información se puede representar de manera más visual con el diagrama de Gantt a continuación (*ver Figura 13*).

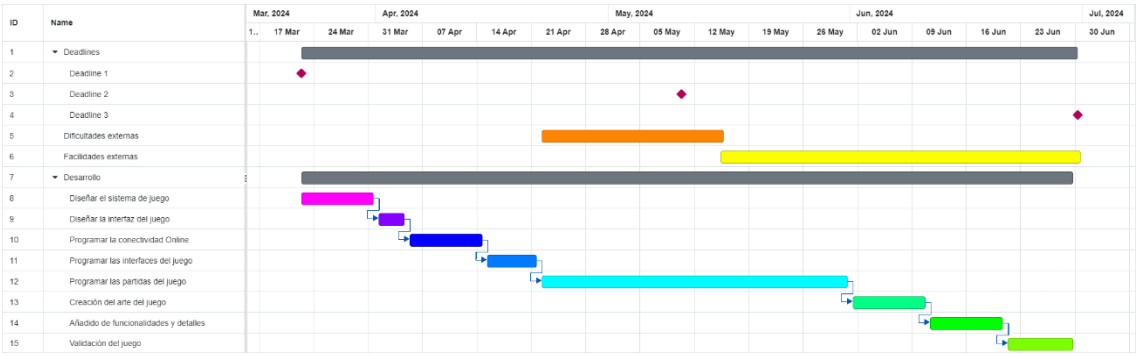

Figura 13. Diagrama de Gantt

(Free Online Gantt Chart Software, s. f.)

3.1.6. Análisis DAFO

A continuación (*ver Figura 14*) se muestra un análisis DAFO con todos los aspectos positivos y negativos del proyecto a tener en cuenta, tanto a nivel interno como aquellos relacionados con el entorno.

Figura 14. Análisis DAFO

(Inicio, s. f.)

3.1.7. Riesgos y plan de contingencias

Como bien se expresa en el análisis DAFO, hay varios aspectos del proyecto que involucran una serie de puntos débiles y riesgos. Con tal de tratar de minimizar el impacto negativo de estos en el proyecto, se ha de tener presente como afrontar esto.

Dependencias con servicios externos. Estos servicios pueden ser limitados, tener un coste de uso y/o dejar de ofrecer soporte para el proyecto en el futuro.	Se intentará minimizar al máximo, dentro de lo posible, el uso de servicios externos de poca fiabilidad o que ofrezcan poca libertad de uso, además de aquellos que implican costes económicos. Para esto, se intentará hacer uso de software *open source.*
Equipo de una sola persona y no tener referentes directos. Esto puede dificultar un buen planteamiento del diseño del juego, además del propio desarrollo a nivel de programación. Esto implica no llegar a unos resultados satisfactorios.	Se intentarán aplicar distintos puntos de vista a los problemas, tanto a nivel de diseño como de programación, con tal de tener una mente abierta y llegar a un resultado óptimo. Además, este resultado se irá testeando durante el desarrollo.

Algunos problemas que surjan pueden consumir más tiempo del previsto.	Las resoluciones de problemas se intentarán planificar, priorizar y adaptar a unos tiempos realistas.
El hardware puede sufrir daños o experimentar un mal funcionamiento.	Se buscarán reemplazos la antes posible.

Tabla 7. Riesgos del desarrollo

También pueden surgir algunos problemas de cara a un producto final, si bien es cierto que no afectan directamente al desarrollo del prototipo.

El género o temática del videojuego puede no ser muy atractiva para un público grande actualmente.	Se deberá dar a conocer el juego a aquellos grupos demográficos que sí sientan interés, como aficionados al ajedrez o a ciertos juegos de cartas. Adicionalmente, es común que estas personas formen parte de grupos o comunidades que pueden ser aprovechados para visibilizar aún más el juego.
Se puede generar una comunidad negativa o “tóxica” en torno al producto.	Esto, aunque no es muy probable teniendo en cuenta la demografía a la que se dirige el juego, puede controlarse de distintas formas, ya sea dentro del mismo juego (*bans*, eliminación de comentarios ofensivos, restricciones de comunicación) como fuera de este (moderando comentarios en redes sociales o enfatizando un tono distinto de cara a la comunicación con los usuarios). De todas maneras, es algo que puede ser mucho más controlado si se toman estas medidas desde el inicio.
Puede lanzarse al mercado un producto que actúe como competencia directa del juego.	Esto se podría aprovechar para analizar las diferencias entre los productos y diferenciarse más potenciando ciertos aspectos que se crean mejores o más convenientes en nuestra propuesta.

Tabla 8. Riesgos a futuro

3.2. Seguimiento

El proyecto ha avanzado correctamente según lo previsto.

Se han podido llevar a cabo algunas pruebas y bases a nivel de programación y arte antes del periodo de tiempo correspondiente.

Algunas herramientas han sido actualizadas a versiones más nuevas debido a que estas ofrecen mejoras y que la documentación de las API abarca estas últimas.

Durante la programación de la conectividad entre jugadores, se planteó la implementación un sistema de *host migration*, es decir, que si el *host* se desconecta, otro cliente en el *lobby* asumiría el rol en su lugar. Fish-Net no soporta esta funcionalidad directamente, por lo que tras varios intentos sin éxito, se ha decidido prescindir de esto, ya que no es algo muy importante para el proyecto.

Finalmente, hay algunos elementos dentro del prototipo con funcionalidades que no han sido programadas. Estos han sido añadidos como *placeholders* porque es una buena práctica tenerlos en cuenta como si de un producto final y completo se tratara, pero debido a su poca o nula relevancia para los fines de este proyecto, la prioridad de añadir sus funcionalidades ha sido la más baja. Como resultado, el prototipo no dispone de efectos de sonido, música, modo espectador, chat funcional ni registro de acciones.

Aunque estas funcionalidades no tienen realmente impacto para los objetivos que se buscan, es importante hacer esta aclaración.

4. Metodología

4.1. Desarrollo

Los objetivos a alcanzar son los que se pueden ver en la siguiente imagen (*ver Figura 15*), en ese mismo orden.

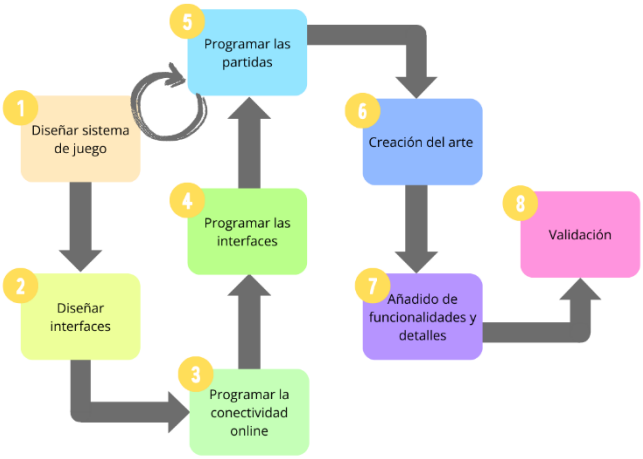

Figura 15. Desarrollo de las tareas

(*Inicio*, s. f.)

Él método de desarrollo que se seguirá será mayoritariamente un desarrollo *en cascada* (o *secuencial*), ya que a niveles generales se irá avanzando a través de una serie de etapas bastante definidas y diferenciadas entre sí.

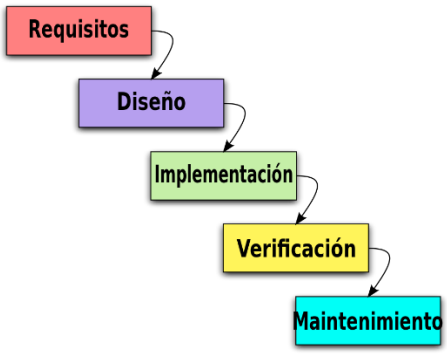

Figura 16. Esquema de desarrollo en cascada

(«Desarrollo en cascada», 2024)

Aún así, cabe mencionar que en cada etapa de desarrollo se intentarán llevar a cabo varias iteraciones hasta que se obtenga un buen resultado. Esto será especialmente

verdad en el caso de la etapa de *Programar las partidas*, ya que, además de las iteraciones que se puedan hacer para corregir *bugs*, se podrán llevar a cabo algunas iteraciones en cuanto al diseño de las partidas con tal de mejorar la experiencia de juego. Esto es debido a que en esta etapa será posible *testear* el juego y sacar algunas conclusiones más *reales* en cuanto a la ya mencionada experiencia de juego. Por esa parte, se puede considerar también una metodología parcialmente ágil.

A continuación, se comenta brevemente en qué consiste cada etapa del desarrollo.

4.1.1. Diseñar el sistema de juego

Se hace una lluvia de ideas sobre distintos elementos que se pueden incluir en las partidas y su funcionamiento. Esto se acompaña de distintos bocetos que ilustren las ideas de una manera más visual. Se seleccionan las que mejor se pueden adaptar al juego y se itera en cada una de ellas hasta que estén lo suficientemente definidas. Luego se valora de nuevo cuáles pueden incorporarse teniendo en cuenta las demás e ideas y cómo coexistirían.

4.1.2. Diseñar la interfaz del juego

El procedimiento es muy similar al del diseño del sistema de juego. Se hace una lluvia de ideas de interfaces, se dibujan bocetos y se itera. En este caso, solo puede quedar una idea final (de cada “pantalla”/interfaz), aunque esta puede contener elementos adaptados de otras. Además, se debe tener en cuenta tanto la imagen de identidad del producto (paleta de colores, formas, tipografías, etc.) como ofrecer la máxima comodidad al usuario.

4.1.3. Programar la conectividad online

Se programa la estructura de código necesaria para que dos instancias del prototipo se conecten de manera remota a través de internet, desde dispositivos y lugares distintos. Para que las comunicaciones entre las instancias funcionen correctamente, se deberá elegir un modelo de conexiones adecuado con los niveles de autoridad pertinentes en cada instancia.

4.1.4. Programar las interfaces del juego

Se debe programar todo el *game loop* sin tener en cuenta la partida en sí. Esto implica que el usuario debería poder iniciar el juego, conectarse a la red, acceder a todos los

menús que haya, poder “ver” otros usuarios o interactuar con ellos, empezar una partida con ellos y salir de esta y volver al menú inicial.

4.1.5. Programar las partidas del juego

Se programa toda la partida entre los jugadores de acuerdo a lo diseñado anteriormente. Se incorporan todas las mecánicas y se conectan con el funcionamiento de la partida.

4.1.6. Creación del arte del juego

Se crea o consigue todo el arte del juego y se aplica, ya que anteriormente se estaba trabajando con *wireframes* y *placeholders*.

4.1.7. Añadido de funcionalidades y detalles

Se añaden posibles funcionalidades menos relevantes, como un chat o algún tipo de personalización dentro del juego. También se perfeccionan cosas que habían quedado en un estado muy simple.

4.1.8. Validación del juego

Se valora el resultado final del prototipo. Para ello, se analizan ciertos aspectos de este con *playtesting* y opiniones externas. Al finalizar, se obtendrán conclusiones sobre posibles mejoras en el juego.

4.2. Herramientas

4.2.1. Visual Studio Community 2022 v.17.10.2

(*Visual Studio 2022 Community Edition*, s. f.)

Entorno de desarrollo integrado que permite editar y ejecutar código de manera eficiente y cómoda. Sus características de *debug* son especialmente útiles. Es gratuito.

4.2.1. GitHub

(*Build Software Better, Together*, s. f.)

Página web que permite guardar archivos (especialmente aquellos relacionados con código) en una nube e irlos actualizando. Mantiene un historial con todos los cambios y

permite crear distintas ramificaciones de proyectos. Resulta particularmente útil para trabajar desde distintos dispositivos, ya sea una única persona o un grupo. Es gratuito.

4.2.2. GitHub Desktop v.3.3.10

(*GitHub Desktop*, s. f.)

Aplicación de escritorio de GitHub para trabajar de una forma más cómoda. Es gratuito.

4.2.3. Unity Editor v.2022.3.3f1

(*Plataforma de desarrollo en tiempo real de Unity | Motor de 3D, 2D, VR y AR*, s. f.)

Motor de videojuegos muy popular y relativamente accesible para usuarios de distintos niveles. Destaca por su gran comunidad y el contenido en internet sobre este, además de su versatilidad para crear distintos tipos de juegos. Tiene un plan gratuito siempre que los ingresos no superen los 100.000$ en los últimos 12 meses.

4.2.4. Unity Hub v.3.8.0

(*Plataforma de desarrollo en tiempo real de Unity | Motor de 3D, 2D, VR y AR*, s. f.)

Administrador de proyectos de Unity y versiones de Unity Editor instaladas. Es gratuito.

4.2.5. Fish-Net v4.2.2

(Introduction, s. f.)

Framework que proporciona varios recursos para la conexión y comunicación de dos instancias de proyectos de Unity. Uno de sus planes es *open source* y gratuito.

4.2.6. Steamworks.NET v20.2.0

(*Steamworks.NET - Steamworks.NET*, s. f.)

Biblioteca que adapta las funcionalidades de Steamworks (set de herramientas de desarrollo de Valve Corporation) a un entorno de C# como el de Unity. Estas herramientas conectan el proyecto con los servicios de Steam. Uno de esos servicios ofrece la conectividad *online* entre varios programas/juegos y es la que se usará principalmente en este proyecto. Es *open source* y gratuito.

4.2.7. Heathen's Steamworks v3.2.1

(Introduction, s. f.)

Toolkit que ofrece facilidades y recursos extra a la hora de usar Steamworks en un proyecto, en este caso uno de Unity. Uno de sus planes es *open source* y gratuito.

4.2.8. FishySteamworks v4.1.0

(FirstGearGames, 2021/2024)

Recurso aplicable a un proyecto de Unity que conecta la implementación de Steamworks (o Steamworks.NET) con Fish-Net. Es *open source* y gratuito.

4.2.9. SandBoxie-Plus v1.12.9

(*Sandboxie-Plus | Open Source Sandbox-Based Isolation Software*, s. f.)

Aplicación que simula un entorno independiente al entorno principal del dispositivo para ejecutar programas, similar a una máquina virtual. En este caso se usará para ejecutar el prototipo a través de una cuenta de Steam distinta (en adición a la cuenta "principal" que ya esté siendo usada por un prototipo en ese momento), pero usando un solo ordenador. Es *open source* y gratuito.

4.2.10. Steam

(*Steam, The Ultimate Online Game Platform*, s. f.)

Plataforma de distribución digital de juegos muy popular (aunque también distribuye otro tipo de programas), desarrollada por Valve Corporation. Sus usuarios disponen de una cuenta que se puede conectar al software durante su uso. Su uso y la creación de sus cuentas es gratuita, pero publicar un software y añadirlo a su catálogo público tiene un coste de 100 $. Este coste sería reembolsado en el caso de que ese software generase una cantidad igual o superior a 1.000 $.

4.2.11. Stable Diffusion

(*Stability AI Image Models*, s. f.)

Modelo de inteligencia artificial de *Stability AI*[27] usado para generar imágenes. Hay varias versiones disponibles, las cuales se han usado para entrenar distintos modelos. Es *open source* y gratuito.

4.2.12. ComfyUI v2.7.2

(comfyanonymous, 2023/2024)

Programa que permite el uso de Stable Diffusion a través de una interfaz visual. Es *open source* y gratuito.

4.2.13. Civitai

(*Civitai*, s. f.)

Página web con una gran cantidad de modelos y demás recursos para la generación de imágenes con IA. Es gratuito.

4.2.13. Figma

(*Figma*, s. f.)

Página web con herramientas muy útiles para realizar pruebas de diseño e interfaces, así como crear iconos y demás elementos sin demasiada complejidad. Uno de sus planes es gratuito.

5. Desarrollo del proyecto

El juego recibirá provisionalmente el nombre de “Karcha”. Es un nombre simple y corto, originado como variación de la pronunciación de “Card Chess” (Ajedrez de cartas).

5.1. Diseño del sistema de juego

El sistema de juego es la base para todo lo demás. Antes de empezar a idearlo, se deben tener en mente algunos aspectos básicos del juego, como qué es lo que se quiere tratar, qué experiencia se le quiere ofrecer al jugador, desde un enfoque muy básico.

[27] Compañía dedicada a la IA generativa y open source.

5.1.1. Elementos básicos

En este caso, como bien se ha especificado anteriormente, se trata de un videojuego competitivo que enfrenta a dos jugadores en un 1 vs 1. Este debe permitir a los jugadores demostrar su habilidad de una forma bastante transparente y justa; igualitaria, similar al ajedrez.

Como este, debe mostrarse simple a primera vista, sin que el jugador tenga que aprender una gran cantidad de reglas o información para empezar a jugar. Así pues, la complejidad debe residir en las interacciones y posibilidades que ofrezcan los elementos, los cuales deben ser, dentro de lo posible, conceptualmente simples y no muchos. Como diría Keith Burgun, el juego debe ser “profundo”, pero no “amplio”; “elegante” (Keith Burgun Games, 2014).

Por otro lado, se le debe ofrecer al jugador una experiencia de juego más “completa” que en el ajedrez, que lo distancie de ser una competición de “ver más lejos” y que les pueda resultar atractivo a jugadores que quieran divertirse. Las partidas deben contener variedad y sorprender en cierto modo a los jugadores, no deben de ser tan “lineales” o “rígidas”. Como se ha indicado con anterioridad, un sistema de cartas puede lograr esto.

Una vez se tiene esto en mente, se pueden definir algunos elementos clave que formarán parte del juego y como afectarían a la partida en rasgos generales.

- Tablero básico con casillas, muy similar al ajedrez, pero con un número de casillas que puede diferir de este, pudiendo ser par o impar.

- Piezas más dinámicas que en el ajedrez, que no solo puedan realizar más acciones, si no que vayan evolucionando durante la partida, tanto variando sus características individuales como la cantidad de piezas en el tablero.

- Sistema de cartas aleatorias que apoye las jugadas con las piezas y que sea lo suficientemente importante como para influenciar la manera en la que los jugadores deberán enfocar sus estrategias, pero siempre respetando en cierto modo al adversario y el determinismo de la partida.

- Sistema de economía que le dé al jugador herramientas para enfocar su estrategia de distintas maneras, permitiéndole invertir en lo que considera más

valioso u oportuno. A la vez, añadir distintos costes a los recursos es una manera fácil y directa de darle un equilibrio el juego.

5.1.2. Ampliación de ideas

Después, una lluvia de ideas permite profundizar más en estos elementos y sus características y usos, o por lo menos explorar diferentes posibilidades y direcciones del juego, antes de analizarlo todo exhaustivamente. Cabe mencionar que algunas ideas pueden ser diametralmente opuestas o incluso contradictorias, pero eso se tendrá en cuenta posteriormente.

Estos son algunos ejemplos, suponiendo que la economía se representa con “monedas de oro” y las fichas/piezas son “unidades”:

- Cada turno se recibe oro.
- El segundo puede empezar con alguna ventaja.
- El oro se puede acumular por turnos.
- El oro puede servir para comprar unidades o cartas o ambas.
- El oro puede servir también para que las unidades hagan acciones como moverse o atacar.
- Se puede añadir otro tipo de “moneda” distinta para las acciones de las unidades, como “energía”.
- La primera acción que realiza una unidad en un turno es gratis o incluso obligatoria de hacer.
- Las unidades tienen un turno de espera al ser invocadas al tablero. En este no pueden realizar ninguna acción.
- Puede haber distintos tipos de unidades con distintos precios.
- Puede haber distintos tipos de cartas con distintos precios.
- Las unidades pueden ganar nivel/valor eliminando a otras.
- Las unidades pueden perder valor al eliminar a otras o hacer acciones.

- Cuantas más unidades tenga, el jugador recibirá más oro (se refuerza invertir en unidades).

- Cuantas más unidades tenga, el jugador recibirá menos oro (se evita un crecimiento exponencial y demasiado ventajoso).

- Los jugadores pueden disponer de una gran cantidad de oro/unidades para empezar.

- Los jugadores pueden empezar sin oro/unidades e ir ganándolo durante la partida.

- Se pueden comprar mejoras para las unidades.

- Las cartas pueden ser el principal o único modo de llevar a cabo acciones con las unidades.

- Las cartas pueden tener condiciones muy específicas para poderse usar, como tener a tres unidades con cierta característica en fila.

- Las características de las unidades pueden variar al usar cartas en esas unidades.

- Las unidades pueden defenderse activa o pasivamente.

- Las unidades pueden tener tipos con ventajas y desventajas sobre otros, como elementos.

- Puede que cada turno el jugador deba invocar una unidad.

- Se pueden vender unidades/cartas por oro.

- Se puede intercambiar cierto número de cartas por otra nueva.

- El uso de cartas puede tener consecuencias que perjudiquen al jugador en cierto aspecto.

- Las unidades pueden tener distintos rangos y direcciones de ataque, movimiento o defensa dependiendo de las cartas que se usen.

- Las cartas o unidades pueden tener efectos activos/inmediatos o pasivos.

- Las unidades que realizan una acción pueden perder la habilidad de usar esa acción de nuevo.

- Las unidades pueden tener ciertos puntos de vida/ataque e incluso defensa.

5.1.3. Definición de ideas

Tras analizar en estas ideas y tener en cuenta en lo que puede derivar cada una, se puede llevar a cabo una mayor definición del juego. Algunas ideas tienen una buena sinergia entre ellas. Otras son opuestas, por lo que se deben descartar aquellas que no encajan tan bien con la dirección en la que se quiere llevar el juego. Otras deben ser modificadas o incluso se pueden combinar distintas ideas para obtener una que esté en un término medio.

Además, se puede aprovechar para definir algunos valores como los costes económicos (dentro del juego) de distintos recursos, lo que ayuda a tener una mejor idea de la importancia de esos elementos y de qué maneras podrán ser usados por los jugadores (en este caso, en sus turnos), a pesar de que es posible que se deban ajustar mejor después de testear el juego.

Una vez este proceso ha concluido, después de varias iteraciones, se obtiene el siguiente resultado, el cual se explicará por secciones:

General

Se trata de un juego 1 vs 1 por turnos. Al inicio de la partida, los jugadores deciden una casilla donde situar su primera unidad. A partir de entonces, estos pueden añadir más unidades durante la partida, entre otras cosas.
El objetivo del juego es acabar con todas las unidades enemigas.

Tablero

El tablero es de 9x9 casillas. Es un poco más espacioso que un tablero de ajedrez debido a que se pretende ofrecer un poco más de “libertad” para maniobrar con los movimientos variados de las fichas.

Oro

El oro será la única moneda del juego y servirá para usar una gran variedad de recursos por distintos precios. Estos son: robar carta especial (3), robar carta de rango (2), robar carta de estadística (3), invocar unidad (5), aumentar la vida de una unidad (2-10, progresivo por nivel de vida), usar una carta en una unidad (1), que una unidad aliada realice una acción (primera acción por turno gratis, independientemente de qué unidad la realice, luego 2), ascender nivel de acción de unidad (5 para nivel 1, 10 para nivel 2, 15 para nivel 3, 20 para nivel 4). Además, las cartas pueden venderse por 1 de oro cada una.

Cada turno, los jugadores recibirán 10 de oro. Es acumulable, por lo que los jugadores pueden llevar a cabo estrategias donde reserven oro durante su turno para usarlo en el futuro. Al mismo tiempo, con tal de limitar un uso excesivo de acciones en un mismo turno, hay un límite de oro total de 50 por jugador.

El segundo jugador puede empezar con algo más de oro (a determinar). Esto es para compensar que juega “después”, pero no debe superar los 5 (+50% del total por turno), ya que en el ajedrez esta desventaja es apenas notoria y en este juego, al añadir un pequeño factor de aleatoriedad, esa desventaja queda aún más disipada.

Cartas

Las cartas afectan al juego de una manera bastante indirecta, dejando el resto a la destreza de los jugadores. Además, sus efectos tienen un valor muy variable dependiendo de la situación en la que los jugadores las usen y cómo lo hagan.

El jugador puede robar las cartas de unos mazos y guardarlas en su mano hasta que vea el momento de usarlas. La mano tiene un máximo de 10 cartas acumulables. Estas no se usan en la partida directamente, sino que aplican sus efectos a las unidades del tablero, y estas son las que ejercen los efectos.

Hay tres tipos de cartas, cada una correspondiente a un mazo distinto.

- **Cartas de rango:**

Estas cartas pueden ser de varios tipos: invocación, movimiento, ataque o defensa. Estas cartas se pueden aplicar a las unidades y les otorgan la capacidad de efectuar

acciones básicas durante sus futuros turnos. Cada carta determina las casillas que pueden ser afectadas por su acción. Por ejemplo, una carta de acción de movimiento puede permitir a una unidad moverse a la segunda casilla que tiene enfrente, pero otra puede indicar la tercera de su derecha. Lo mismo ocurre con los otros tipos de acciones.

Además, las cartas de invocación contienen un tipo de unidad distinto (terrestre, acuático o aéreo) que será el que se le aplique a la nueva unidad invocada.

- **Cartas de estadística:**

Estas cartas pueden ser de dos tipos: de ataque o de defensa. Las cartas de ataque tienen un valor numérico entre 1 y 10, ambos incluidos, mientras que las de defensa lo tienen entre 1 y 5, ambos incluidos. Estos valores se usarán en los combates entre unidades. Además, estas cartas también tendrán un tipo de combate incluidas. Así pues, al equiparse en una unidad, esta recibirá ese tipo de combate, además del valor numérico de ataque o defensa.

- **Cartas especiales:**

Estas cartas pueden ser de dos tipos: pasivas o activas. Ambos tipos se aplican a las unidades del tablero, pero las pasivas tienen un efecto que está presente en todo momento mientras están equipadas a una unidad, mientras que las activas le ofrecen a la unidad la posibilidad a realizar una acción especial. Estas cartas, a diferencia de las de rango o estadística, son muy distintas unas de las otras y tienen efectos muy variados, algunos de los cuales desaparecen al cumplirse ciertas condiciones, como que pasen X turnos (en el caso de las pasivas) o que se use la acción especial X veces (en el caso de las activas).

Algunos ejemplos de pasivas son: “La defensa de esta unidad se iguala a la de una unidad adyacente. En caso de ser varias, estas se suman” o “Esta unidad puede atravesar los bordes del tablero y aparecer por el lado opuesto”.

Algunos ejemplos de activa son: “Convierte una unidad en un rango de 5 casillas en el tipo de combate del usuario”, “Anula una pasiva de una unidad en un rango de 3 casillas” o “Intercambia los puntos de ataque y defensa”.

Unidades

Las unidades son las piezas del tablero, tanto aliadas como enemigas. Estas pueden realizar acciones (invocar, moverse, atacar, defenderse, usar una acción especial o meditar) durante el turno del jugador.

Disponen de puntos de vida. Estos pueden reducirse a lo largo de la partida, generalmente al ser atacados por otras unidades. Si se reducen a 0 o menos, la unidad muere y desaparece del tablero. Si era la última unidad en pie de un jugador, este pierde.

Las unidades empiezan con 1 punto de vida, pero esta cantidad puede aumentarse de uno en uno a cambio de una cantidad de oro igual a la vida aumentada que se obtendrá.

Además, las unidades tienen un tipo de combate concreto. Según este, se clasifican en: terrestres, acuáticas y aéreas. Las terrestres infligen el doble de daño a las acuáticas, las acuáticas lo hacen con las aéreas y las aéreas con las terrestres. Del mismo modo, si las unidades se encuentran en desventaja de tipo, su ataque infligirá la mitad de daño.

Acciones

Una acción es un recurso que el jugador puede usar durante su turno a través de una unidad aliada en el tablero. La primera acción que el jugador realice en un turno es gratuita, mientras que las siguientes cuestan 2 de oro. Al hacer la primera gratis, se incentiva a los jugadores a actuar en cada turno e ir construyendo su estrategia poco a poco, ya que es más rentable que hacer varias acciones de golpe en un mismo turno, aunque esto último ofrece algo más de sorpresa para el rival.

Hay varios tipos de acciones: invocación, mover, atacar, defender, especial y meditar.

Para que una unidad aliada pueda realizar una acción de cualquiera de esos tipos menos “meditar”, el jugador deberá antes equipar una carta de rango de ese tipo en la unidad. Por ejemplo, si el jugador equipa una carta de rango de movimiento en una unidad aliada, podrá moverse con esa unidad a partir de entonces. Las unidades tienen asignado un nivel por cada tipo de acción, empezando por el nivel 0 y llegando al nivel 4 como máximo. Este nivel indica la cantidad de cartas de acción de ese tipo que puede tener equipadas una unidad al mismo tiempo, yendo de 0 a 4. Así pues, si una unidad un nivel de acción de movimiento 3 y 3 cartas de rango de movimiento equipadas, podrá escoger una casilla para moverse de entre 3 opciones.

Sin embargo, las cartas no se pueden aplicar a las unidades en cualquier momento. Para ello, la unidad tiene que estar en un estado de meditación. Este estado lo obtiene después de usar la acción de meditar.

- Meditar:

Cuando una unidad medita, entra en un estado de meditación que perdurará hasta su siguiente turno. Las unidades en meditación no pueden usar ninguna acción y su vida se vuelve 1 temporalmente. Durante este estado, el jugador puede equipar cartas a su unidad. Al turno siguiente la unidad despertará y podrá hacer uso de esas cartas a partir de entonces. La meditación también permite aumentar la salud de la unidad y sus niveles de acción a cambio de oro, así como reemplazar cartas ya equipadas en la unidad.

- Invocar:

La unidad invoca en cierta casilla a una unidad aliada nueva, siempre y cuando la casilla no esté ocupada por otra unidad. La unidad será del tipo de combate especificado en la carta de rango de invocación. Además, tendrá 1 punto de vida y nivel 0 de acción para cada tipo, sin ninguna carta equipada. Hacer una invocación tiene un coste de 5 de oro adicional a los costes de acción.

- Mover:

La unidad cambia su posición a cierta casilla libre. No importan las casillas por las que “pase” para llegar hasta ahí (como un caballo que salta unidades en el ajedrez).

- Atacar:

La unidad compara sus puntos de ataque con la defensa del rival seleccionado, el cual está en cierta casilla, y la unidad enemiga recibe el daño, aunque este puede ser mitigado por defensas directas o de soporte. En caso de atravesar la defensa, el daño será infligido directamente en los puntos de vida de la unidad enemiga. Si su vida se reduce a 0 o menos, la unidad muere.

- Defensa:

Las casillas marcadas en este atributo indican dos cosas:

- Si un enemigo se encuentra en esta casilla al atacar, la unidad puede defenderse del ataque como defensa directa.

- Si una unidad aliada se encuentra en esta casilla al recibir un ataque, la unidad puede apoyarla en su defensa como defensa de soporte.

Al usar la acción, la unidad usa su escudo de forma activa, por lo que sus puntos de defensa se duplican. Sin embargo, las defensas de soporte en las que participe podrán reducir sus puntos de defensa o incluso romper por completo su escudo. Además, la unidad no podrá hacer uso de sus otras acciones, como atacar o moverse.

Desactivar su defensa activa tiene un coste de acción igual al resto de acciones. Al desactivar la defensa, los puntos de defensa de la unidad se reducen a la mitad. En caso de que el resultado no sea exacto, se redondeará hacia abajo.

- Acción especial:

La unidad llevará a cabo los efectos indicados en su carta de acción especial activa. Los efectos son muy variados y puede que tengan un límite de usos o se deban cumplir ciertas condiciones.

Combate

Cuando una unidad enemiga A ataca a una unidad aliada B, se deben enfrentar los puntos de ataque de A con los de defensa de B en esa situación. Los puntos de ataque de A son los que esa unidad dispone según su carta de estadística de ataque equipada. Los puntos de defensa de B son 0 inicialmente, pero pueden incrementarse según ciertas condiciones:

- B tiene una carta de rango de defensa equipada. Esa carta tiene como casilla objetivo la casilla en la que se encuentra A. En ese caso, se añaden a la defensa los puntos de defensa indicados en la carta de estadística de defensa equipada. Se trata de una defensa directa.

- Una unidad aliada de B tiene una carta de rango de defensa equipada. Esa carta tiene como casilla objetivo la casilla en la que se encuentra B. En ese caso, se añaden a la defensa los puntos de defensa indicados en la carta de estadística

de defensa equipada en esa unidad. Se trata de una defensa de soporte (o *support*).

Además, si una de las unidades aliadas usó la acción de defender y su defensa se duplicó, se trata de una defensa activa.

En el caso de realizar una defensa directa o una defensa *support* activa, los escudos de esas unidades van al frente del ataque y son vulnerables. Eso significa que el ataque reducirá los puntos de defensa permanentemente (a no ser que se equipe otra carta de estadística de defensa en la unidad más adelante). El primer escudo en reducir sus puntos de defensa es el que está realizando la defensa directa. En caso de que sus puntos de defensa lleguen a 0, este se romperá y pasarán a defender las siguientes defensas activas, en orden de activación (uso de la acción de defender de la unidad), rompiéndose cada una de igual manera si los puntos de ataque restantes (ya que el ataque es mitigado por cada defensa anterior) dejan en 0 a la defensa.

Poniendo un ejemplo práctico, se muestra la siguiente situación.

Hay un ataque enemigo de A (ataque 10) contra B (defensa 4 y vida 3). C (defensa 1), D (defensa 3) y E (defensa 5) son unidades aliadas. En turnos anteriores, se ha usado la acción de defender en C y E, por lo que ellos están usando defensas activas y realmente sus puntos de defensa actuales son de 2 y 10, respectivamente.

- La defensa de B marca la posición de A, por lo que B hace una defensa directa de 4.
- La defensa de C marca la posición de B, por lo que C hace una defensa *support* activa de 2.
- La defensa de D marca la posición de B, por lo que D hace una defensa *support* inactiva de 3.
- La defensa de E no marca la posición de B, por lo que no apoya su defensa de ningún modo.

Así pues, la defensa resultante es 4 (B) + 2 (C) + 3 (D) = 9, respecto los 10 puntos de ataque.
Primero se romperá el escudo de B y quedará un daño restante de 10 – 4 = 6.
Después, se romperá el escudo de C y quedará un daño restante de 6 – 2 = 4.
Después, se mitigará daño con la defensa de D, pero su escudo no se verá reducido ni se romperá, porque no es una defensa activa, y quedará un daño restante de 4 – 3 = 1.

Ese punto de daño será aplicado a B, así que su vida bajará de 3 a 2.

Si en vez de esto, D hubiese usado una defensa activa, sus puntos serían 6 en vez de 3, por lo que la última operación hubiese sido 4 – 6 = –2. El daño habría sido completamente mitigado y B no habría perdido puntos de vida, pero la defensa de D se habría reducido de 6 a 2 permanentemente.

El combate puede parecer complejo, pero su funcionamiento está ideado para incentivar jugadas estratégicas, como planificar qué unidades poner en defensa de otras y hasta el orden en el que se hace, con tal de parar un futuro ataque enemigo. O, como atacante, plantearse qué unidades es más rentable atacar, ya que derrocar primero a algunos soportes sin defensa propia puede facilitar el ataque al objetivo inicial. Aun así, se pueden producir varios cambios en el futuro, especialmente tras el testeo.

5.1.4. Cambios realizados tras iterar

A continuación se mencionan algunos conceptos que han sido cambiados tras varias iteraciones del diseño del juego, ya que en el apartado anterior solo se mencionan los resultados a los que se ha llegado.

- Antes, el rango de valores defensivos era 1 – 10 y las defensas de soporte pasivas daban la mitad de ese valor. Se cambió para ofrecer cálculos menos complicados para los jugadores.

- Antes, las cartas de rango y de estadística eran un solo tipo de carta que ofrecía ambas características. Se cambió para simplificar los cálculos, ya que distintos rangos en una misma unidad tenían distintos valores de ataque/defensa y eso los complicaba excesivamente.

- Antes, no había estado de meditación, por lo que las unidades podían actuar con sus nuevas acciones en el mismo turno que las obtenían, lo cual podía sentirse como un *output randomness* injusto para el contrincante.

- Antes, las defensas activas no inutilizaban la unidad y terminaban al siguiente turno. Se cambió para incentivar más a los jugadores a mantener estas defensas activas durante varios turnos como medio de protección por posibles ataques futuros, sin tener que hacer grandes suposiciones para decidir si era rentable o no gastar oro en defenderse ese turno y los siguientes. Esto también implica que los jugadores pueden usar unidades de un modo más defensivo, pero dejando de lado su ofensiva como compensación. Además, el desactivar la defensa activa de una unidad tiene un coste, lo que permite a los jugadores trazar estrategias teniendo eso en cuenta, tanto en el caso de unidades del propio jugador como las del oponente. Por último, el poder activar y desactivar la defensa de una unidad en el mismo turno ofrece la posibilidad de cambiar el orden de los escudos que defienden una casilla.

- Antes, se podía aumentar el ataque y la defensa con oro. Esto haría que los jugadores dejaran de usar las cartas de estadística, lo cual le da variedad y renovación a la partida.

5.1.5. Ideas que no se descartan a futuro

Las siguientes ideas son posibles medidas a tomar en caso de que el juego se testee y no funcione bien a nivel de diseño o se reconozcan posibles mejoras.

- Simplificar las cartas especiales. Esto puede hacerse haciendo que todas sean activas.

- Cambiar el sistema de puntos de ataque o defensa o tipos de combate incluidos en las cartas de estadística.

- Simplificar el combate. Aunque se puede ayudar al jugador mostrando la información de una manera muy visual, el juego no debería suponer un gran esfuerzo a nivel de cálculos matemáticos.

- Limitar el número de niveles de acción máximo a menos de 4 o empezar con 1 en vez de 0.

- Si los rangos de valores de las cartas de estadística de ataque y defensa se igualaran, se podrían unir ambos tipos de cartas en uno solo darle la libertad al jugador de escoger si desea utilizar el valor de la carta en ataque o defensa.

Incluso si los rangos no se igualan, se le podría dar al jugador la posibilidad de usar una carta de defensa como ataque, a pesar de que el rango sea menor.

- Asignar un valor de ataque/defensa por carta de rango equipada en una unidad, en vez de por unidad, lo que añade complejidad.

- Adjuntar los tipos de combate en las cartas de estadística de ataque solamente, o crear cartas específicas de tipo de combate, las cuales formarían parte del mismo mazo de cartas de estadística de ataque y defensa.

- Limitar las máximas unidades en combate a 10 por jugador y/o, de alguna forma, evitar un exceso de invocaciones.

- Incentivar de alguna manera a los jugadores a exponerse y atacarse en vez de quedarse en su bando aumentando el poder de sus unidades, así como en el ajedrez es obligatorio mover una pieza en cada turno.

- Que la acción de defender otorgue 5 de defensa y al desactivarse reste otros 5, en vez de duplicar y reducir a la mitad.

- Aunque tener un nivel de acción distinto para cada acción fomenta hacer una buena gestión de oro y gastarlo solo en lo necesario, se puede plantear la posibilidad de que solo exista un único nivel por unidad. Este puede permitir cierta cantidad de cartas equipadas en cada acción o incluso indicar esa cantidad a nivel global, combinando todos los tipos de acciones.

5.2. Diseño de la interfaz del juego

La interfaz del juego ha de ser simple. Mientras que los menús serán “abstractos”, la interfaz de la partida tendrá una ambientación de juego de mesa de fantasía medieval que representa una guerra entre dos ejércitos.

5.2.1. Identidad visual

Colores

Se usará la combinación de distintos tonos de verde y amarillo para crear su identidad visual, ya que tienen un buen contraste.

Verde:

Las tonalidades de verde constituyen un espectro cromático muy amplio, por lo que distintos tonos pueden usarse fácilmente para indicar distintos elementos, y aun así ofrecer el suficiente contraste como para que cada uno de ellos sea distintivo. Además, el verde puede asociarse con la tranquilidad, aprendizaje y desarrollo intelectual, lo que se puede relacionar con el tipo de juego.

Amarillo:

Por el contrario, el amarillo no ofrece un espectro cromático muy amplio, por lo que su uso puede quedar limitado a ciertos elementos con características similares. En este caso, se usará para botones con los que el usuario puede interactuar. De esta forma, el usuario distinguirá fácilmente estos elementos tan importantes en su navegación, debido a que seguirán este patrón establecido. Además, el amarillo puede asociarse con la energía, lo que rompe con lo “estático” del verde para indicar fácilmente estos elementos interactivos.

En general, la combinación de estos dos colores es práctica y crea un buen equilibrio visual para la interfaz de los menús.

Formas

Se usarán formas simples y bastante geométricas. Por lo general se hará bastante uso de rectángulos, aunque no se descarta un acabado más “redondeado” al momento de pulir el arte del juego y pasarlo a limpio. Los iconos también se mantienen simples. En este punto la mayoría han sido específicamente creados para el proyecto, pero hay algunos que se usan como *placeholders* temporales.

Tipografía

Se han sopesado bastantes tipografías con estilos variados. Aquellas sin serifas eran más adecuadas para los menús más extradiegéticos y lo que requería una buena legibilidad. Por el contrario, la ambientación medieval de la partida sintonizaba mejor con una letra con más detalles y serifas pronunciadas. Finalmente, se escogió la tipografía “Amaranth” como punto medio entre los dos conceptos, aunque siempre dando prioridad a una buena legibilidad.

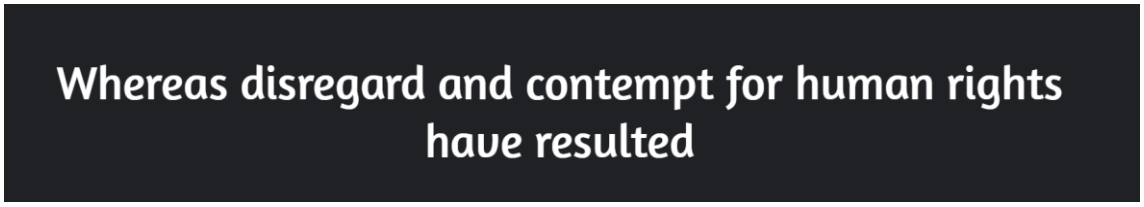

Figura 17. Ejemplo de uso de la tipografía Amaranth

Se aprecian pequeñas serifas y formas curvas que le dan al texto un carácter alegre, pero manteniendo la legibilidad del texto.

(*Amaranth*, s. f.)

De todas maneras, también se recurrirá a la tipografía “Nugie Romantic” para algunos detalles, los cuales no serán textos muy largos y reforzarán la ambientación de las partidas en un contexto de fantasía medieval. Un ejemplo de ello serán los nombres de las unidades del tablero.

Figura 18. Ejemplo de uso de la tipografía Nugie Romantic

Se aprecian serifas y detalles muy pronunciados, lo que le da al texto un ambiente fantasioso y, hasta cierto punto, reminiscente de la escritura a mano con tinta característica de una ambientación medieval.

*(*Nugie Romantic Font | dafont.com*, s. f.)*

5.2.2. Pantalla de inicio

Figura 19. Diseño de la pantalla de inicio y paleta de colores

La pantalla de inicio es muy simple y solo consta de un botón para conectarse como elemento principal, situado en el centro para otorgarle esa importancia. El juego debe estar conectado a Steam para que el botón de conectar funcione.

5.2.3. Pantalla de selección de lobbies

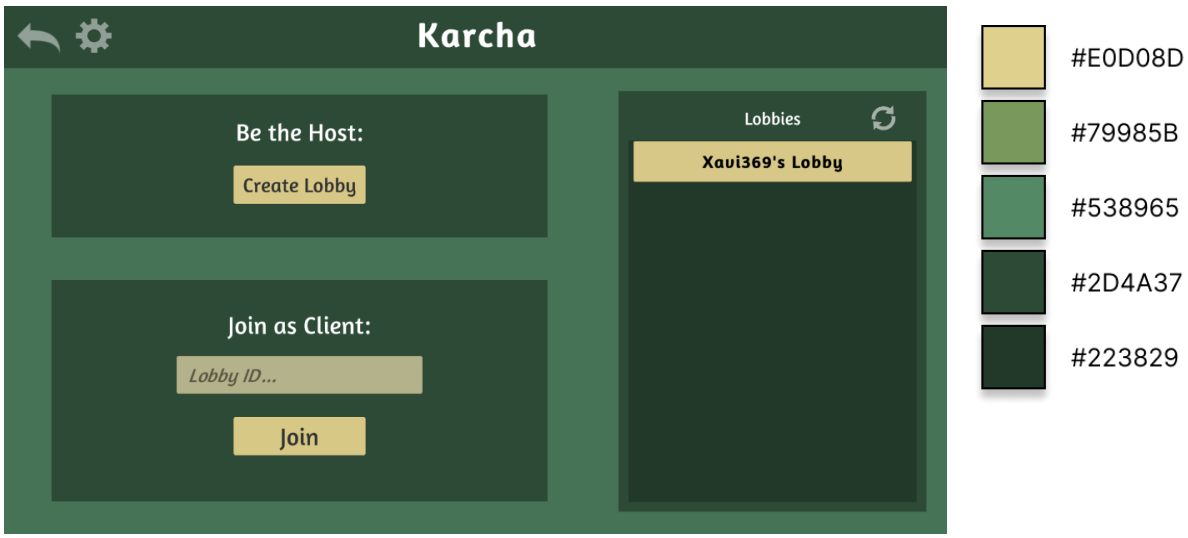

Figura 20. Diseño de la pantalla de selección de lobbies y paleta de colores

La pantalla de selección de *lobbies* mantiene una estética parecida pero algo más "seria", con colores menos saturados. Se distinguen tres secciones principales:

- Crear *lobby*: Al hacer clic en el botón, el usuario creará un *lobby* y será el *host* de la sala.
- Unirse a un *lobby* por código: El jugador puede escribir el código identificativo del *lobby* al que quiere unirse y apretar el botón. Si el código es válido, se conectará al *host* y se unirá al *lobby* como cliente.
- Lista de *lobbies*: Un listado que muestra los *lobbies* activos actualmente. Se actualiza con frecuencia, pero también hay un botón para refrescar la lista manualmente de forma instantánea. Al hacer clic en uno de los *lobbies* mostrados, el jugador se conectará al *host* y se unirá como cliente.

5.2.4. Pantalla de lobby

Figura 21. Diseño de la pantalla de lobby

Esta interfaz mantiene la paleta de colores anterior. Una vez el jugador está en el *lobby*, puede ver el nombre de la sala, su código identificativo (lobby ID), una lista de los jugadores presentes y unos botones en la parte inferior. Los jugadores muestran el nombre e imagen de su perfil de Steam, un texto que indica si están listos o no para empezar la partida y unos iconos que simbolizan si el jugador formará parte del bando azul o rojo en la partida, o si por el contrario será un espectador. Además, el *host* tiene un icono de una corona que lo distingue del resto.

Los jugadores pueden cambiar su papel en la partida haciendo clic en su perfil. Además, el *host* puede hacer clic en los perfiles del resto y expulsarlos del *lobby*.

También pueden cambiar su estado de “listo” o “no listo” a través del botón “Set Not Ready”/“Set Ready”. El botón “Leave” sirve para salir del *lobby* y volver al menú anterior. Por último, el botón “Start Game” sirve para empezar la partida. Este aparecerá en la pantalla del *host* cuando haya un único jugador tanto en el bando azul como en el rojo y todos los presentes en el *lobby* estén listos.

5.2.5. Pantalla de partida

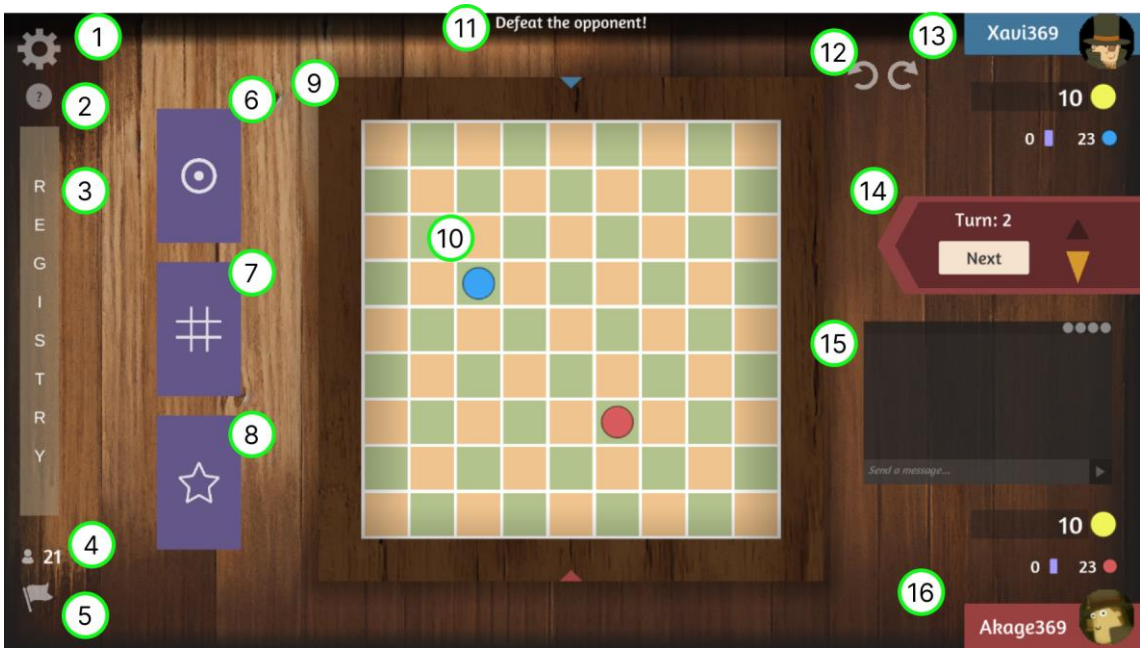

Figura 22. Diseño de la pantalla de partida con indicaciones

En este ejemplo, el jugador local juega en el bando rojo, mientras que el enemigo en el azul.

En esta pantalla los jugadores del bando azul y rojo tendrán su duelo. Los elementos de la interfaz, por lo general, intentan mantener una estética acorde con la fantasía medieval.

Se identifican los siguientes elementos:

1. Icono para acceder al menú de ajustes.
2. Icono para acceder al menú de ayuda/instrucciones.
3. Listado de eventos que van ocurriendo durante la partida, en orden.
4. Icono para acceder al menú de jugadores en la partida (incluyendo espectadores).
5. Icono para rendirse.
6. Mazo de cartas de estadística.
7. Mazo de cartas de rango.
8. Mazo de cartas especiales.
9. Tablero de juego.
10. Unidad/pieza de juego.
11. Espacio de instrucciones y otros textos.
12. Iconos para rotar el tablero en 90 grados en las distintas direcciones.

13. Perfil del jugador enemigo. Cerca de su nombre e imagen se encuentra su información de la partida, siendo esta el número de oro que tiene, cartas en mano y unidades desplegadas en el tablero.
14. Información sobre el turno actual. Contiene un botón para que el jugador termine su turno.
15. Chat de la partida.
16. Perfil del jugador local. Contiene la misma información que el del enemigo.

Además, la mano de cartas de cada jugador se mostrará en su bando del tablero; la del enemigo arriba y la del jugador local abajo.

Por último, las unidades del tablero se muestran como fichas circulares simples, del color identificativo del jugador en cuestión. Sin embargo, estas unidades contienen información. Esta esta se mostrará si el jugador la selecciona pasando su ratón por encima (o la unidad se mantiene seleccionada si el jugador hace clic en ella).
Esta información se mostrará en lado izquierdo del tablero, opacando los mazos de cartas.

Estos son algunas variantes de ejemplo:

Figura 23. Variantes de unidad seleccionada con indicaciones

Las unidades de arriba pertenecen al bando azul, mientras que las de abajo al bando rojo. En cada bando se distinguen los diferentes tipos de combate. Estos son, de derecha a izquierda, acuático, aéreo y terrestre. La unidad azul de la izquierda tiene nivel 4 de acciones y está equipada con 3 cartas en cada una. Las demás tienen nivel 1 y una carta equipada.

Se identifican los siguientes elementos:

1. Vida.
2. Rangos de invocación.
3. Rangos de movilidad.
4. Imagen.
5. Nombre.
6. Rangos de ataque.
7. Rangos de defensa.
8. Acciones especiales. Las que tienen fondo blanco son activas, y las que no, pasivas.
9. Estadística de ataque.

10. Estadística de defensa.
11. Tipo de combate.

Cabe mencionar que si el jugador mantiene su cursor sobre algunos de los elementos de la unidad, se mostrará más información sobre ese elemento. Por ejemplo, si el cursor está sobre el nombre de la carta especial equipada en la unidad, se mostrará al lado esa carta y su descripción. Si el cursor está sobre el icono de ataque, defensa, invocación o movimiento, se mostrará en el tablero las casillas correspondientes afectadas por ello.

Las cartas tendrán un tipo de interacción muy parecida a las unidades. Si el ratón pasa por encima de una carta de la mano, su información se hará visible de la misma manera en la que lo hacen las unidades. Si se hace clic en esa carta, la carta quedará seleccionada y su información se mostrará hasta que se deseleccione.

Además, tanto en el caso de las unidades como en el de las cartas, si el elemento está seleccionado, aparecerán debajo de esta varios botones con varias opciones de interacción disponibles.

5.2.6. Pantalla de fin de partida

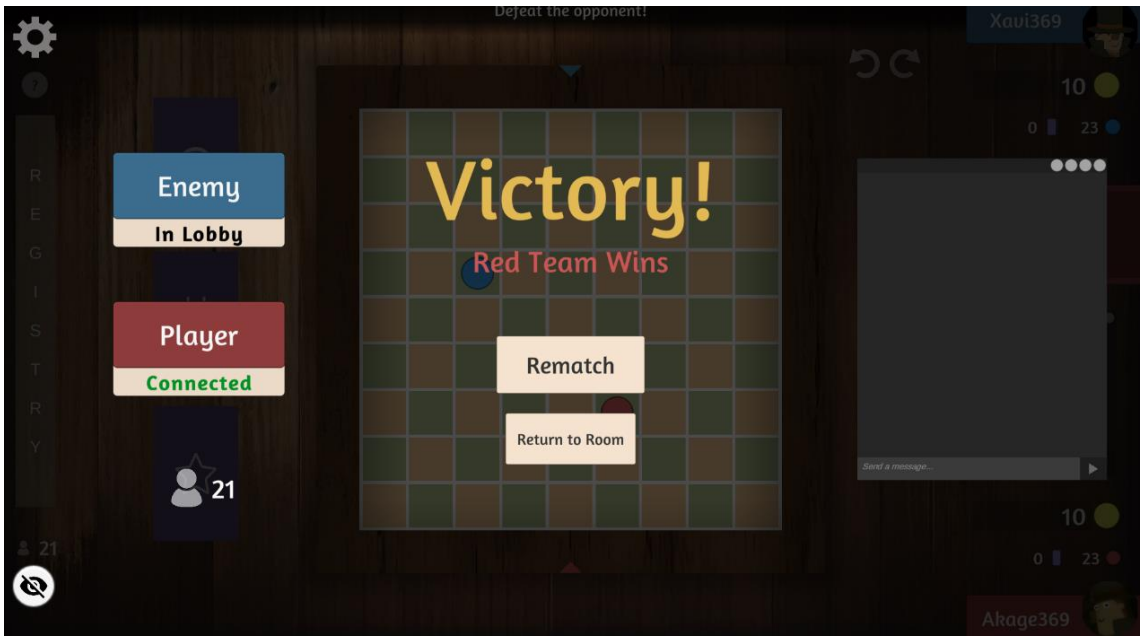

Figura 24. Pantalla de fin de partida

La pantalla de fin de partida es simple. Aparece por encima de la pantalla de partida, mientras que esta queda sombreada. En caso de querer ver el estado de la partida de

nuevo, el jugador puede hacer clic en el botón del ojo de la esquina inferior izquierda, y volver a hacer clic luego para retornar a esta pantalla.

A parte del icono de ajustes, el resto de elementos de la izquierda son dos casillas que indican el estado de los jugadores azul y rojo (conectados, esperando otra partida, en el *lobby* o desconectados) y el icono para ver la lista de jugadores de la partida. Por otra parte, en el lado derecho se encuentra el chat. En el centro, el texto indicando quien ha ganado y dos botones, uno para volver a empezar la partida y otro para volver a la pantalla de *lobby*.

5.3. Programación de la conectividad online

A continuación se indica el procedimiento llevado a cabo en las distintas escenas para gestionar la conexión entre jugadores, siendo la primera escena la escena “offline”.
Cabe mencionar que se nombrarán varios términos propios de Unity y el lenguaje de programación C#.

5.3.1. Escena offline

Set-Up Inicial

Lo primero a configurar es un *game object* llamado “NetworkManager”. Este se encargará de iniciar y mantener todo el sistema de conexión entre varias instancias del juego; o lo que es lo mismo, varios jugadores. Este *game object* tendrá varios componentes pertenecientes al *framework* Fish-Net mencionado con anterioridad y, de hecho, será muy parecido al propio prefab de demostración con el mismo nombre que viene incluido en el *framework*.

El mayor cambio realizado respecto a esta configuración por defecto es el añadido del componente “FishySteamworks”. Este se añade al campo “Transport” del componente “TransportManager” y a partir de entonces se usará como enlace entre las funcionalidades de Fish-Net y Steamworks. Es decir, Fish-Net usará los servidores de Steam como “canal” de conexión. También es importante marcar el campo “Peer To Peer” del componente como “true”, ya que en este juego las conexiones serán de ese tipo; los jugadores no se conectarán a un servidor externo, sino que se conectarán entre ellos, siendo Steam solamente el “enlace” entre jugadores que permite a cada uno saber la IP y demás información del resto.

De todas formas, con tal de crear grupos de jugadores conectados entre sí (llamados *lobbies*), uno de los jugadores deberá tomar el papel *host* y, en cierta parte, el rol de "servidor". Será el jugador central, "líder" y "autoridad" del grupo. Será quien mantenga a todos conectados. En caso de que un jugador pierda la conexión con él, este abandonará el *lobby*. En caso de que el *host* pierda su conexión, el resto de jugadores lo hará también.

Sin embargo, antes de conectarse con Fish-Net y con el resto de jugadores, cada jugador debe, individualmente, conectarse con Steam, ya que si no, no será posible que este haga uso Steamworks y se conecte con el resto.

Conexión con Steam

Para conectar el juego con Steam, específicamente con la cuenta de Steam activa en el ordenador en ese momento, basta con activar un *script* llamado "SteamworksBehaviour" de Heathen's Steamworks. En este caso, cuando el usuario hace clic en el botón "Connect!" de la pantalla inicial, se activa un *game object* llamado "BootstrapManager", el cual activa a su vez el componente SteamworksBehaviour que tiene asociado.

Después, al obtener el resultado de este intento de conexión, se activa un evento con este, el cual llama a una función. Esta función comprueba si el resultado es positivo o negativo.

Si es negativo, se muestra un mensaje de error en pantalla pidiendo al usuario que abra Steam y BootstrapManager se desactiva, esperando a ser activado de nuevo más adelante y repetir el proceso.

Si es positivo, se avanza a la siguiente escena ("Global"). Además, se obtiene el Steam ID y *username* del perfil del jugador (a través de las funciones **SteamUser.GetSteamID()** y SteamFriends.GetPersonaName(), respectivamente). El Steam ID del jugador será especialmente útil en el futuro, ya que servirá para identificar cada jugador y su instancia de juego o conexión. Estos datos se guardarán en una nueva instancia de una clase llamada "PlayerData", siendo la instancia "localPlayerData". Esta instancia se guardará en un *scriptable object* que también es un *singleton*, llamado "ConnectionManager". También cabe mencionar que tanto los objetos de BootstrapManager como NetworkManager no se destruirán al cambiar de escena, ya que se seguirán usando posteriormente.

Otro detalle a mencionar es que al componente SteamworksBehaviour se le debe asociar una configuración. Esa configuración es un objeto de tipo *scriptable object*. Este objeto tiene varios campos modificables a tener en cuenta en caso de querer lanzar el juego a Steam de manera oficial. Uno de los más importantes es el campo de Application Id.

Application Id

Este es un número identificativo único para cada juego publicado en Steam. Una de sus funciones en esta configuración es tenerlo en cuenta a la hora de conectarse con otros perfiles de Steam en el juego, ya que estos perfiles deben estar jugando al mismo juego. Así pues, la conexión entre jugadores solo tendrá en cuenta instancias de juego con ese mismo número id asociado. Como este número solo se puede obtener publicando un juego en Steam oficialmente y esto conlleva un costo económico, el desarrollo de este prototipo se llevará a cabo con un Application Id público y común para todas aquellas aplicaciones que no están publicadas en Steam oficialmente. Este es 480. De este modo los jugadores de este prototipo podrán conectarse entre sí, aunque también podrán visualizar y conectarse con otros perfiles externos que también estén usando este Application id desde otras aplicaciones distintas. Esto, lógicamente, debe ser evitado, ya que provocaría errores en el funcionamiento. Además, el uso de este Application Id también tiene como consecuencia que el juego sea percibido como “Spacewar” desde la interfaz de Steam.

5.3.2. Escena global

La escena global es la que permite a los jugadores conectados con su cuenta de Steam conectarse a otros jugadores. Para esto, se ha optado por ofrecer tres posibilidades: crear un *lobby*, unirse a un *lobby* a través de un código directo y visualizar una lista de *lobbies* activos. Una gran parte del funcionamiento de esto ocurre en el *script* de BootstrapManager. En este *script*, se asocian unas funciones a unos *callbacks* de Steamworks. Estos *callbacks* se activan cada vez que ocurre alguna acción específica indicada en ese *callback*. Algunos ejemplos importantes son:

- **LobbyCreated_t:** El usuario crea un *lobby*.
- **GameLobbyJoinRequested_t:** El usuario pide unirse a un *lobby*.
- **LobbyEnter_t :** El usuario se une a un *lobby*.

- **LobbyMatchList_t:** El usuario obtiene una lista de *lobbies*.

- **LobbyDataUpdate_t:** El usuario obtiene información actualizada de los *lobbies*.

- **LobbyChatUpdate_t:** El usuario obtiene información sobre un miembro del *lobby* que ha entrado o salido de este.

Cuando los *callbacks* se activan, las funciones asociadas usan su información para actuar en consecuencia de distintas maneras. Algunos de estos se activan automáticamente cuando el usuario está conectado a un *lobby*, como ocurre en el caso de LobbyChatUpdate_t, pero otros lo hacen solo como respuesta a unas peticiones del usuario, como en el caso de solicitar crear un *lobby*, unirse a uno u obtener una lista de estos y su información. Esto es lo que ocurre en los tres casos mencionados anteriormente:

Crear un lobby

Para crear un *lobby* basta con hacer uso de la función **SteamMatchmaking.CreateLobby()** de Steamworks.NET. En esta se añaden como parámetros el número máximo de jugadores del *lobby* y el tipo de *lobby*, pudiendo ser este público, privado, solo para amigos, etc. Usando el *callback* LobbyCreated_t, el usuario obtendrá el Lobby ID del nuevo *lobby* y lo puede guardar en una variable. Además, también definirá una serie de características para el *lobby* con varias llamadas a la función **SteamMatchmaking.SetLobbyData()**. Esta función permite tres parámetros.

1. Lobby ID del *lobby* al cual se añadirá la información nueva.
2. String que servirá como key identificatoria, como ocurre con un diccionario.
3. String que servirá como valor, como ocurre con un diccionario.

Gracias a esto se le puede añadir información básica a un *lobby*, como un nombre, el Steam ID del *host* y una palabra o código clave que sirva para identificar que se trata de un *lobby* de esta aplicación en concreto, a modo de diferenciarse de otros *lobbies* creados en otras aplicaciones pero que también comparten el Application Id 480. Tanto este código como cualquier otra información añadida a un *lobby* puede ser usada más adelante como filtro a la hora de obtener una lista de *lobbies* que cumplan ciertos requisitos.

Además, después de eso, se asigna el Steam ID del usuario al campo correspondiente del componente FishySteamworks del NetworkManager. Este campo es “Client Address” y será la identificación del usuario a nivel de conexión de red de Fish-Net, la cual indicará a quién se deben conectar los usuarios. En este caso, al ser el *host*, se pone el Steam ID del propio usuario. En el caso de los demás clientes, estos pondrán el Steam ID del *host*.

Después, el *host* activa su conexión con Fish-Net con el método de FishySteamworks **StartConnection()**, adjuntando un parámetro “true” para activar su rol de servidor.

Si un usuario crea un *lobby* con éxito, su siguiente acción será unirse a este como *host*, de forma automática, y activará el *callback* LobbyEnter_t de la misma manera que lo harán los demás clientes que se unan en el futuro.

Unirse a un lobby a través de un código

Unirse a un *lobby* a través de su código es tan sencillo como crearlo. De nuevo, basta con una única función de Steamworks.NET: **SteamMatchmaking.JoinLobby()**, la cual necesita el Lobby ID como parámetro. Después, a través del *callback* LobbyEnter_t, el usuario puede obtener información sobre el *lobby* de una manera muy similar a como se añade esta al crearlo; a través de la función **SteamMatchmaking.GetLobbyData()**. Esta necesita el Lobby ID y *key* y devuelve el valor asociado.

Después, de igual modo que el *host* hacía al crear el *lobby*, se establece el Steam ID del *host* en el componente FishySteamworks y se inicia la conexión de Fish-Net con **StartConnection()**, aunque esta vez con “false” para activar el rol de cliente sin más.

Visualizar una lista de lobbies

La lista de *lobbies* se crea a partir de varios prefabs, cada uno representando un *lobby* activo distinto. Estos prefabs contienen información de los *lobbies* como su nombre o Lobby ID, por lo que el usuario es capaz de hacer clic en estos y unirse al *lobby* a través de esa Lobby ID, de igual manera que lo harían escribiendo la ID como código en el apartado anterior.

Siempre que la lista sea visible, la información de los *lobbies* activos se obtiene de forma continua cada X intervalos de tiempo, usando un *coroutine*. Esta información se obtiene primero añadiendo un filtro de cantidad máxima de *lobbies* que se quieren obtener. Esto

es gracias a la función de Steamworks.Net **SteamMatchmaking.AddRequestLobbyListResultCountFilter()** y añadiendo un número como parámetro.

Luego, se añade un filtro que solo admite aquellos resultados que tengan un valor A asociado a una *key* B, siendo A igual a un valor que sea especificado. En este caso, esto es usado con el código identificativo del prototipo de juego que anteriormente fue asociado a los *lobbies* creados en este. La función es la siguiente: **SteamMatchmaking.AddRequestLobbyListStringFilter()**, pasando como parámetros la key, valor deseado y el tipo de comparación a hacer, en este caso que sea igual (ELobbyComparison.k_ELobbyComparisonEqual).

Después, **SteamMatchmaking.RequestLobbyList()** inicia la petición de búsqueda de *lobbies*. El resultado se obtiene con el *callback* LobbyMatchList_t. Se itera en la lista de *lobbies* obtenida. Por cada *lobby*, se obtiene su Lobby ID con **SteamMatchmaking.GetLobbyByIndex()** y su índice como parámetro y se llama a **SteamMatchmaking.RequestLobbyData()** con el Lobby ID como parámetro para obtener información detallada sobre el *lobby*. Esa información se obtendrá con el *callback* LobbyDataUpdate_t. Entonces, se comprobará si ese *lobby* ya existe en la lista o no, en cuyo caso se instanciará un prefab nuevo con su información.

Además de esto, también se usan algunos *callbacks* propios de Fish-Net. Estos se usan en ConnectionManager. Algunos de los más importantes son aquellos que se activan cuando el usuario activa o desactiva su conexión. Cuando ocurre lo primero, se activa la interfaz de “Lobby” y se aplica a esta la información correspondiente, como el nombre. Cuando ocurre lo segundo, esta se desactiva y se retorna al menú global.

Menú de lobby

El menú de *lobby* se activa cuando los usuarios se conectan a un *lobby* de forma satisfactoria. Al mismo tiempo, cuando un usuario empieza su conexión con Fish-Net, el componente “PlayerSpawner” de NetworkManager instanciará de forma automática un prefab que representará ese cliente. Este tiene el nombre de Client y tiene un componente “NetworkObject” asociado. Este componente es esencial para las conexiones con Fish-Net y permite varias cosas, dos de las cuales son especialmente importantes:

- Hacer llamadas a funciones en las demás instancias del juego y recibir esas llamadas desde esas otras instancias. Estas llamadas son conocidas como Remote Procedure Calls o RPCs y es en lo que se basa la interacción entre varios clientes.

- "Spawnear" otros *network objects* (objetos con el componente NetworkObject) dentro de "networked scenes". Las *networked scenes* son escenas que están sincronizadas en la red y, por tanto, permiten a los *network objects* que hay dentro de ellas establecer conexión entre ellos y mandarse mensajes como lo son las RPCs. Por ejemplo, si el juego fuese un *shooter* en el que los jugadores se mueven se mueven por un mapa e interactúan con el entorno, abriendo puertas, moviendo o destruyendo cajas, etc., entonces el mapa tendría que cargarse como una *networked scene* y esos elementos serían *network objects*. De ese modo, todos esos elementos estarían sincronizados entre las distintas instancias del juego. *Spawnear* un *network object* en una *networked scene* no es más que "añadirlo" a esa escena para que esté sincronizado con todos los clientes. Como se ha mencionado, solo un *network object* puede *spawnear* otro. Además, ese *network object* debe pertenecer al servidor (o *host*) para poder hacerlo. Los *network object* también tienen la característica de tener un dueño o "owner" del cual son propiedad. Ese *owner* se le es asignado al *network object* cuando este es *spawneado*, aunque un *network object* perteneciente al servidor puede cambiar el *owner* de cualquier otro *network object* más adelante.

Así pues, cuando un cliente se conecta a un *lobby*, el PlayerSpawner de NetworkManager instanciará en la escena un *network object* Client automáticamente. En ese momento, Client también hará *spawn* en la escena activa actual (la cual será convertida en una *networked scene* si no lo era) y recibirá como *owner* al usuario de esa instancia de juego. Esto ocurrirá con todos los clientes a medida que se conecten al *lobby*. Sin embargo, esto se ha adaptado para el prototipo.

En este caso, se le ha añadido un *script* Client al prefab Client. Se trata de un *script* derivado de NetworkBehaviour, lo que significa que tiene todas las funcionalidades de un *game object* típico de Unity (Monobehaviour), pero con propiedades de NetworkObject añadidas, las cuales se establecen automáticamente en el *script* de acorde a las propiedades del componente NetworkBehaviour asociado al mismo *game object*. Cuando el prefab se instancia en la escena, el *script* Client activa su método **Start()**. En este se comprueba si el cliente es el servidor con la propiedad

NetworkBehaviour.IsServerStarted, siendo “true” un resultado afirmativo. En ese caso, se instanciará otro prefab de *network object* llamado “OnlineManager” (con el método común **Instantiate()**) y se *spawneará* en la escena con el método **ServerManager.Spawn()**.

Después, cuando OnlineManager esté correctamente inicializado, se activará su método *callback* **OnStartClient()** propio de los NetworkBehaviour, del cual se hace un *override* para añadir más código. Se vuelve a comprobar si se trata del servidor y, en caso afirmativo, se crea y se carga una nueva *networked scene* llamada “Online” y se configura de tal forma que los próximos clientes en conectarse *spawnearán* sus *network objects* Client en esa escena. Además, se transfieren de escena tanto el Client del servidor como el OnlineManager. Así pues, a partir de entonces, todos los *network objects* de los clientes (Client) estarán en la escena Online, además de otro *network object* llamado OnlineManager que pertenece al servidor.

OnlineManager contendrá todos los métodos referentes a la conexión entre clientes como las RPCs, además de información sobre todos los clientes conectados al *lobby* y, más adelante, su información en la partida.

La información de los clientes es almacenada en un diccionario. Sus elementos tienen el Steam ID del cliente como *key* y una instancia de la clase PlayerData anteriormente mencionada como valor. Esta clase tiene toda la información que se pueda necesitar de cada cliente, desde su nombre de usuario hasta si están listos o no para empezar la partida y en qué bando quieren estar en esta.

Realmente no se trata de un diccionario común, sino de un SyncDictionary. Este es un tipo de variable propio de Fish-Net que estará sincronizado con todos los clientes. Así pues, cada usuario, desde su instancia del juego, puede acceder a OnlineManager y obtener el valor de SyncDictionary en cualquier momento, y si este ha sido modificado desde el servidor, el usuario obtendrá el valor modificado.

Una característica importante es que los valores de SyncDictionary, así como otros tipos de variables sincronizadas, solo pueden ser modificados por el servidor, por lo que si un cliente quiere modificar un valor, este tiene que enviar un RPC al servidor indicando el cambio. El servidor lo recibirá y hará el cambio. Después, los demás clientes recibirán esos cambios en la variable. Esto ofrece más seguridad al juego, ya que se necesita la autoridad del servidor.

Además, las variables sincronizadas también ofrecen *callbacks* que se activan en los clientes cada vez que una variable actualiza su valor. Es muy importante hacer uso de estas para asegurarse de que las variables han sido modificadas antes de intentar hacer alguna acción que pueda dar lugar a errores al usarse un valor antiguo que no ha sido actualizado a tiempo en todos los clientes.

Algo a añadir es que en muchas ocasiones se activará el *callback* conforme un valor ha sido modificado, pero no se tendrá ninguna certeza sobre cuál es ese valor. Así pues, puede resultar conveniente el uso de booleanas que lo indiquen cada vez que se realiza la modificación. Por ejemplo, en vez de cambiar solamente si un cliente está listo o no para empezar la partida, se puede combinar el cambio con poner en “true” una booleana “setReadyForGameHasChanged”. Después, en la función *callback* se hacen comprobaciones sobre si alguna de estas booleanas es “true” y si es el caso, se determina que ese valor ha sido modificado y se pueden realizar acciones en consecuencia, además de volver a establecer la booleana en “false”.

Gracias a todos estos recursos, el menú de *lobby* es capaz de mostrar una lista de con todos jugadores del *lobby* y su información, lo cual está indicado en el SyncDictionary de OnlineManager en forma de elementos PlayerData. Estos elementos se añaden al diccionario cada vez que un cliente se conecta al *lobby*, específicamente en la función *override* **OnStartClient()** del OnlineManager anteriormente mencionada, la cual se activa localmente en el cliente que se conecta. Entonces, se manda un RPC con la información del jugador en forma de PlayerData, siendo parte de esta información que ya ha sido establecida anteriormente, por ejemplo al conectarse a Steam, y enviada a la variable localPlayerData anteriormente mencionada. A partir de ese momento, si se quiere seguir haciendo uso de localPlayerData para acceder cómodamente a la información del cliente local, este debe estar siempre actualizado y a la par con la última información de su variante en el diccionario de OnlineManager. Para esto, cada vez que se modifica la información de un PlayerData en el diccionario, se usa el *callback* del cambio para enviar a su cliente la nueva versión actualizada, a través de un RPC.

Además, cuando se añade un PlayerData nuevo a la lista, se comprueban los jugadores actuales y su rol en la partida (bando azul, bando rojo o espectador) y se determina el rol que debería tomar el nuevo jugador automáticamente (si ya existe el bando azul, el nuevo jugará como rojo y viceversa), aunque cada jugador puede cambiarlo más adelante.

Adicionalmente, el *host* también tiene la opción de expulsar jugadores del *lobby* haciendo clic en el botón “Kick”. Esto se hará obteniendo el Steam ID de ese jugador (el cual también es su ConnectionAddress de Fish-Net) y guardándolo como la variable ulong “playerDataSteamId”. Entonces, se obtendrá su conexión de Fish-Net (llamada NetworkConnection) a través de la siguiente línea de código:

```
NetworkConnection conn =
NetworkManager.ServerManager.Clients.ToList().Find(x =>
ulong.Parse(x.Value.GetAddress()) == playerDataSteamId).Value;
```

Y luego se llamará a la función de Fish-Net **ServerManager.Kick()** con esa conexión como parámetro.

Finalmente, si existe un bando azul y rojo y todos los jugadores están listos, el *host* tiene la opción de empezar partida, lo cual mandará a todos los jugadores a descargar la escena actual a través de un RPC. El servidor comprobará si todos lo han hecho a partir de unas booleanas en sus PlayerData. Si ese es el caso, un nuevo RPC les hará cargar la escena de juego. De nuevo, el servidor comprobará si todos los clientes lo han hecho, y, de ser así, la partida empezará. Esta se llevará a cabo, como es habitual, a través de RPCs.

5.4. Programación de las interfaces del juego

Debido a que las interfaces del juego son bastante simples, no hay mucho por comentar, ya que la complejidad reside en el funcionamiento a nivel de conexiones, el cual ha sido explicado en el apartado anterior.

5.4.1. Game loop

El *game loop* se ha logrado realizar correctamente. El usuario es capaz de moverse entre pantallas sin limitaciones. Basta con apretar el botón de “Connect!” para avanzar a la escena global y entrar a un *lobby* para ver su menú y tener la posibilidad de entrar en partida. Una vez en partida, se puede realizar el proceso opuesto rindiéndose con el icono de la bandera y apretando “Return to lobby”. Una vez en la interfaz de *lobby*, el usuario puede hacer clic en el botón “Leave” para volver al menú global y el icono de la fecha hacia atrás para regresar a la pantalla de inicio.

5.4.2. Desconexión con Steam

Un dato a tener en cuenta es que la librería de Steamworks está pensada para estar enlazada en todo momento con el juego, por lo que no contiene funcionalidades para "desconectarse" de este. Así pues, una vez Steam está activo y el jugador ha hecho clic en "Connect!" y se ha conectado satisfactoriamente con su cuenta, en el caso de que esta conexión se pierda (cerrando Steam, por ejemplo) mientras la aplicación sigue ejecutándose, esta se cerrará automáticamente al intentar realizar cualquier acción que implique el uso de Steamworks.

Esto incluye el caso en que el usuario, después de conectarse, regresa al menú inicial y cierra Steam. Cuando apriete el botón, no saldrá un mensaje de error en pantalla como haría normalmente, sino que la aplicación se cerrará automáticamente. Esto es debido a la comprobación que se realiza en ese momento, la cual da por hecho que Steam, al haber sido conectado con anterioridad, lo sigue estando.

5.5. Programación de las partidas del juego

A continuación se explica la lógica de cómo se han logrado llevar a cabo algunas de las funcionalidades más destacables de la partida.

5.5.1. Información de la partida

La información de la partida se almacenará en una variable de una clase llamada "GameStateInfo". Esta será una variable sincronizada (syncVar) de OnlineManager que seguirá la misma metodología usada en el resto de variables sincronizadas, explicada anteriormente. Esta clase contiene información como el número de turno actual (int), el índice del equipo activo durante este turno (int), el estado de fin de juego activo (es un enum con las opciones "None", "WinTeam1", "WinTeam2" y "Draw") y una lista de equipos. Estos equipos son esencialmente los jugadores enfrentados en la partida, y se muestran en forma de clase "Team". Esta clase contiene información como el oro de cada jugador (int) y un diccionario llamado "units" con un número ID (*key*) e información sobre la unidad del tablero asociada a este (valor).

Hay que tener en cuenta que estos campos, que deben estar sincronizados entre clientes, no deben contener variables derivadas de algunas clases de Unity directamente, como los Monobehaviour, ya que estos objetos pueden dar problemas al transferirse entre clientes distintos. Así pues, se debe hacer una clara distinción entre

estos elementos locales y aquellos de clases más básicas o *custom classes* que no dan problemas al transferirse.

Un ejemplo de ello son las unidades. Como estas tienen una parte de lógica muy relacionada con los *game objects* del tablero que las representan, esta lógica no se transmite entre clientes y se aplica en un *script* Monobehaviour en los *game objects*, llamado “Unit”. Por otro lado, la información global de la unidad como su posición, nombre u otras características se establece en una *custom class* llamada “UnitData”, la cual sí que es segura de transmitir entre clientes. Esta es la clase que se asocia como valor al diccionario anteriormente mencionado. Por otra parte, los *scripts* Unit tienen un campo “referenceUnitDataID” el cual hace referencia al ID de la unidad, el cual es la *key* del diccionario de los valores UnitData. De esta manera, ambos campos quedan relacionados.

5.5.2. Tablero

Renderizado

Con tal de hacer pruebas y modificaciones de una manera fácil, el tablero no es una simple imagen dentro del juego. En realidad, se trata de un componente gráfico de UI que se genera en el juego de acorde a unas variables dadas, en una zona designada para ello. El tablero se genera a partir de un *script* derivado de la clase “Graphic” de Unity, haciendo *override* a su función *callback* **OnPopulateMesh()**.

Este *callback* viene con un parámetro de tipo VertexHelper. Este usa sus funciones **AddVertex()** con un vértice como parámetro y **AddTriangle()** con tres índices. Con la primera, se establecen unos vértices o puntos a seguir, mientras que con la segunda esos puntos se unen con líneas que se renderizan en pantalla, siguiendo el orden especificado. Combinando estas opciones, se puede lograr crear una cuadrícula que actúe como tablero.

Por parte de los vértices usados, estos son instancias de la clase UIVertex y usan principalmente dos parámetros, “color” (de tipo Color32) para establecer su color y “position” (de tipo Vector3) para su posición. Una vez configurado el vértice, se añade al VertexHelper con las funciones especificadas anteriormente.

Todo lo demás se basa en cálculos que determinan la posición de cada vértice, teniendo en cuenta la anchura y altura de la zona del tablero, el número de casillas verticales y horizontales y el grosor deseado para el borde del tablero y la separación entre casillas.

Rotación

El tablero se puede rotar para facilitar que el jugador pueda analizar la situación desde otros puntos de vista, como es habitual en el ajedrez. El usuario puede hacerlo apretando los botones correspondientes. Cada botón gira 90 grados el tablero en una dirección u otra, dependiendo del botón. Cada vez que el jugador hace clic, se suman 90 grados más a la rotación, lo que permite al jugador de una manera muy cómoda acumular varias rotaciones haciendo varios clics antes de que la primera haya concluido.

Esto se consigue gracias a que la zona del tablero tiene un *script* donde revisa de forma continuada (en su método **Update()**) si su rotación es igual a los grados que supuestamente debe tener. Si no lo es, la booleana de animación de rotación en esa dirección será “true” y la animación correspondiente para llegar al siguiente estado de rotación del tablero se activará, mientras que si los grados y la rotación coinciden, la booleana será “false”.

Además, el tablero tiene un *script* “Board” que maneja algunas funcionalidades e información de este, como índices de casillas en x e y (casilla (1,1), casilla (1,2), casilla (1,3), etc.) y las posiciones de estas en la pantalla. Esto permite identificar en qué casilla hace clic el jugador, recurso que se usará a menudo durante la partida. Así pues, los índices y posiciones se recalcularán cuando la rotación se detenga (cuando la rotación del tablero coincide con los grados deseados). De esta forma, después de una rotación de 90 grados en el sentido horario, la posición donde antes había la casilla (1, 1) ahora marcará la casilla (1,9), por ejemplo.

Además, cabe mencionar también que este *script* dispone de métodos de conversión entre índices de casillas locales y globales, ya que es más cómodo trabajar con índices locales en cada instancia de juego, pero cuando se necesita información conectada entre clientes, se deben usar índices globales donde, en este caso, la casilla (1,1) está tocando el lado azul y la casilla (1,9) toca el lado rojo, mientras que localmente esto varía dependiendo el bando del jugador.

5.5.3. Selección de casillas en el tablero

Al apretar un botón, se deben poder invocar unidades en el tablero. Luego el jugador debe seleccionar la casilla que quiera y confirmar con otro botón.

El procedimiento empieza detectando que casillas están disponibles para realizar la invocación. En caso de tratarse de la invocación inicial de la primera unidad, simplemente se obtienen los índices de las casillas de cierta área. En caso de hacer la invocación a través de una unidad seleccionada, se obtienen los índices de casillas de invocación disponibles de la unidad, repasando la lista de rangos que tiene asignados (variable “localSquaresCanInvoke”). Estos índices se adaptan con algunos métodos tanto a la orientación de la unidad (dirección en la que “mira”) como a la posición de esta (ya que obviamente la posición final de la casilla depende directamente de la posición donde se encuentra la unidad en ese momento). Estas casillas se añaden a una lista de Vector2Ints llamada “globalSquaresCanInvoke”.

Todo esto sucede en un *script* llamado “BoardManager”. Este *script* tiene tres listas de Vector2Ints: unavailableSquares, availableSquares y selectedSquares. Cada una de estas indicará las casillas a mostrar de cada tipo (no disponibles, disponibles y seleccionadas).

Además, el *script* también tiene asociados tres prefabs, cada uno correspondiente a uno de esos tipos, los cuales son: unavailableInvokeSquarePrefab, availableInvokeSquarePrefab, y selectedInvokeSquarePrefab, respectivamente.

Retomando el proceso, por cada casilla de globalSquaresCanInvoke, se comprueba que ninguna unidad esté situada en esa misma posición o que la casilla quede fuera de los límites del tablero, lo que inhabilitaría la invocación. De darse alguno de esos casos, esta casilla se añadiría a unavailableSquares, mientras que si la casilla está libre se añadirá a availableSquares.

Entonces, por cada casilla en unavailableSquares y availableSquares, se creará un nuevo objeto de tipo InvokeSquare. Esta clase deriva de la clase Square, la cual puede ser utilizada por otros tipos de casilla más adelante, como las casillas para moverse, atacar o defenderse, cada una con sus peculiaridades. InvokeSquare tiene un constructor que necesita un Vector2Int y en enum SquareState (este puede ser “None”, “Available”, “Selected” o “Unavailable”). El primer valor es la propia casilla que se está

revisando, mientras que el segundo depende de la lista en la que se encuentre este valor. Además, el constructor tiene un último parámetro enum que indica el tipo de combate que recibirá la nueva unidad invocada. Este se recupera de localSquaresCanInvoke, donde está especificado.

Entonces, al crear la InvokeSquare, el constructor instanciará un prefab. El prefab depende del SquareState y su posición será aquella que se obtenga a partir del Vector2Int dado, calculándose esto con una función de conversión de Board. Después, el InvokeSquare se añadirá a una lista llamada “selectionSquaresList” para futuras revisiones.

Después, se ordenarán estas casillas para que su visibilidad sea la que les corresponde, y casillas de más abajo del tablero no tapen aquellas de más arriba, ya que hay un elemento de estos prefabs que sobresale sobre la casilla en sí (una flecha).

A partir de entonces, si el jugador hace clic en una casilla perteneciente a availableSquares, esta se eliminará de la lista y se añadirá a la lista de selectedSquares y se repetirá el proceso, esta vez como una casilla seleccionada. En caso de hacer clic en una casilla de selectedSquares, pasará lo contrario y esta será “disponible” de nuevo.

Si hay una casilla seleccionada y el usuario hace clic en el botón “Confirm”, se llamará a un RPC con el índice del equipo que la invoca, índice de su casilla (a nivel global) y la dirección en la que “mira” la unidad (a nivel global) y se eliminarán todas las casillas de selección y se limpiarán las listas.

Al recibir el RPC, el servidor creará una UnitData nueva y le asignará esas características, así como otras como nombre un nombre e imagen, y la incorporará al diccionario del equipo correspondiente, dándole un número ID que se incrementa cada vez que se invoca una unidad nueva. Luego, a nivel local, cada cliente, al recibir ese UnitData, instanciará el prefab de la unidad en el tablero, el cual tiene el *script* Unit, y se le asignará a Unit el ID de la unidad, para que este la tenga como referencia.

Cabe mencionar también que en vez de hacer clic izquierdo sobre una casilla disponible para seleccionarla como opción, el jugador puede hacer clic derecho y efectuar la invocación en esa casilla de forma directa.

La acción de mover una unidad sigue un procedimiento prácticamente igual, solo que el RPC final no llama a una función para crear una nueva unidad, sino para cambiar la posición de una existente. Por otro lado, la acción de atacar también aprovecha la gran mayoría de esta estructura, pero no la funcionalidad de poder seleccionar una casilla o activar algún efecto al hacer clic en esta. En vez de eso, las casillas marcadas solo tienen un propósito visual, pero en cada una se comprueba si una unidad está contenida en ella y, de ser así, se le activa un bool “canBeTarget” a esta, lo que la hace susceptible a ser seleccionada como target; en este caso, de ataque.

5.5.4. Muestra visual de rangos

Algunos conceptos del punto anterior se vuelven a usar a la hora de mostrar visualmente las casillas afectadas por los rangos de distintos tipos de las unidades. Las casillas afectadas se obtienen a partir de las listas de rangos localSquaresCanInvoke, localSquaresCanMove, localSquaresCanAttack o localSquaresCanDefend y las adaptaciones pertinentes como la posición de la unidad “usuaria” y su dirección. Después de obtener estas casillas, se instancian unos prefabs que indican visualmente el tipo de rango que afecta a esa casilla. Todo este proceso se llama cuando el ratón pasa por encima de algunos iconos (estadística de ataque y defensa o los propios iconos de rango de la unidad) o la misma unidad. Cuando el ratón sale de esas zonas, se eliminan los prefabs creados.

5.5.5. Nombre e imágenes de unidades

Las unidades en el tablero deben tener un nombre e imagen para que sean claramente identificables. Esto se mostrará en la interfaz de información de la unidad, juntamente con sus estadísticas. Estos deben ser en la medida de lo posible únicos y, por lo tanto, no repetirse. Esto implica que para asignar estos campos, se debe hacer uso a una lista de nombres e imágenes con una gran cantidad de estos, lo que, especialmente en el caso de las imágenes, puede suponer una carga desmedida de recursos innecesarios que puede afectar al rendimiento del juego.

Por esto, se ha intentado enfocar esta carga de recursos de una manera óptima y no de una manera ineficaz como podría ser que un *game object* cargue más de 1.000 nombres o más de 100 imágenes. En su lugar, se cargarán solo los textos necesarios en cada situación, siendo estos cargados desde la carpeta “Resources” de Unity, que tiene esa finalidad.

Nombres

Hay dos listas de nombres, una con nombres femeninos y otra con nombres masculinos. Cada una de estas listas está un formato simple: un archivo “.txt” con un nombre en cada fila.

Dependiendo del sexo asignado a la unidad, se usará un “path” u otro, en forma de string, el cual conducirá hasta el archivo correspondiente.

Después, se usará el siguiente código para obtener el texto del archivo:

```
TextAsset txtAsset = Resources.Load<TextAsset>(resourcePath);
string text = txtAsset.text;
```

Y con el siguiente código se obtendrá el número de líneas (nombres) del archivo:

```
// Recorrer cada línea
while ((index = text.IndexOf('\n', index + 1)) != -1)
{
	count++;
}

// Incrementar el contador si la última línea si no termina con '\n'
if (text.Length > 0 && text[text.Length - 1] != '\n')
{
	count++;
}
```

Luego, se ejecutará el siguiente código:

```
public void ChooseRandomLineAndSet(int unitId, string filePath, uint lineCount, bool isResource)
{

    SetUsedNames();

    bool hasToRepeat = false;
    bool canRepeatName = false;
    invalidIndexList.Clear();

    do
    {
        uint randomLineIndex = 0;

        do
        {
            randomLineIndex = (uint)Random.Range(0, lineCount);
        }
        while (invalidIndexList.Contains(randomLineIndex) && canRepeatName == false);

        hasToRepeat = CheckRandomLine(unitId, filePath, lineCount, randomLineIndex, ref canRepeatName, isResource);
    }
    while (hasToRepeat);

    unitNamesUsed.Clear();
    invalidIndexList.Clear();
}
```

Figura 25. Código para seleccionar nombres de unidades

En este, se añaden a la lista “unitNamesUsed” los nombres usados por las unidades activas con **SetUsedNames()**. Luego se elige un índice de línea aleatorio. Si este se encuentra en la lista de índices no válidos (“invalidIndexList”), se repite la selección. Si, por el contrario, es un nombre aparentemente válido, se llama a **CheckRandomLine().**

En esta función, se itera varias veces en el texto del archivo hasta obtener los índices de inicio y final de la línea deseada y se substrae el texto con la función **string.Substring()**. Además, **string.Trim()** asegura que el texto esté en un buen formato. Luego, si la variable “canRepeatName” es falsa, comprueba si el nombre es válido y se usará, si no tiene un formato válido (como que esté en blanco) o ya está en uso. Si canRepeatName es “true”, no se tendrá en cuenta esto último.

Si el nombre es correcto y se usa, la función acaba ahí y retorna “false”, por lo que no se repite el bucle. Si no, el índice se añadirá a la lista invalidIndexList. Además, se hará otra comprobación: si el número de elementos en esta lista es igual a al número de líneas del archivo. De ser así, la lista se vaciaría y canRepeatName se volvería “true”, así que a partir de entonces los nombres podrían estar repetidos. Entonces se devolverá “true” y el bucle se repetirá. Esto es porque se supondría que todos los nombres posibles de la lista ya han sido usados, por lo que la alternativa en ese hipotético caso sería usar nombres repetidos.

Una vez se obtiene el nombre, basta con establecerlo en la UnitData de la unidad.

Imágenes

El caso de las imágenes es muy parecido al de los nombres. En este caso, con tal de tener una “lista” de posibles imágenes a elegir con un índice que las identifique, se ha optado por crear primero un archivo “.txt” que contiene en cada línea el nombre de la imagen. De hecho, se crean dos archivos distintos, uno con las imágenes asociadas al sexo masculino y otro con las del sexo femenino. Cada uno de ellos se crean con el siguiente código:

```
public void SetIndexFile(string filePath, string dataPath)
{
    // Verificar si estamos en el editor
    if (!EditorApplication.isPlayingOrWillChangePlaymode)
    {
        // Obtener todos los archivos PNG en la carpeta Resources
        string[] pngFiles = Directory.GetFiles(dataPath, "*.png");

        // Crear un archivo de texto para el índice y escribir los nombres de los archivos PNG
        using (StreamWriter writer = new StreamWriter(filePath))
        {
            foreach (string imagePath in pngFiles)
            {
                // Obtener el nombre del archivo sin la ruta
                string imageName = Path.GetFileNameWithoutExtension(imagePath);
                // Escribir el nombre del archivo en una linea del archivo de indice
                writer.WriteLine(imageName);
            }
        }

        Debug.Log("Archivo de índice creado con éxito en: " + filePath);
    }
}
```

Figura 26. Código para generar un archivo de nombres de imágenes

La variable "filePath" indica la ruta donde se generará el archivo, mientras que "dataPath" indica la ruta de donde se obtienen las imágenes.

Después de esto, la metodología a continuación es prácticamente igual que en el caso de los nombres, aunque el resultado que se obtiene es distinto. Una vez se selecciona satisfactoriamente una línea del archivo que no se ha usado antes, se guardará tanto el índice de esa línea como un string que junta el *path* de la carpeta de la imagen (de nuevo, este parte de Resources, pero varía dependiendo el sexo de la unidad) y el texto de la línea (es decir, el nombre de la imagen).

El índice indicará la línea del archivo usada, mientras que el *path* indicará la ruta de la imagen. El índice se usará principalmente para que se tenga en cuenta que la imagen en esa línea ya está siendo usada, como ocurre con los nombres en su respectiva lista. El *path*, por otro lado, servirá para que el *game object* correspondiente a esa UnitData cargue la imagen y esta pueda mostrarse en la interfaz de información de la unidad en el futuro. Esto se hará a través del mismo método usado para cargar el archivo de nombres, de la siguiente manera:

```
Sprite image = Resources.Load<Sprite>(unitData.imagePath);
```

5.5.6. Selección de unidades

El jugador puede pasar el ratón por encima de una unidad del tablero para ver su información. Esto recibirá el nombre de "soft selection". Una vez en ese estado, si el jugador hace clic en la unidad, esta pasará a un estado de "info selection", donde además de ver la información de la unidad, el jugador también verá las opciones

disponibles que tiene esa unidad para realizar en ese momento, como sus acciones (mover, atacar, etc.). Además, si una unidad está seleccionada como “info”, el jugador puede apartar el puntero de la unidad, pero seguirá viendo su información y opciones hasta que no deseleccione la unidad, lo cual puede llevar a cabo haciendo clic en esta de nuevo, apretando el botón “Deselect” o seleccionando otra unidad.

Para lograr esto, se siguen una serie de pasos. Antes de eso, cabe mencionar que las unidades del tablero son *game objects* a los cuales se les ha asignado una zona cuadrada como zona interactiva con el puntero del ratón. Esta es una variable Rect con las mismas dimensiones que una casilla del tablero, las cuales han sido calculadas con anterioridad.

El primer paso es obtener la posición del puntero con el siguiente código:

```
mousePos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
```

Esto se lleva a cabo en un una función en **Update()**. Después, otra función en **Update()** revisa si la zona interactiva de la unidad contiene esa posición con el método **Rect.Contains()**. Si el resultado es “true”, la unidad se unirá a una lista “unitOnMouseList” si no está dentro ya, y si es “false” la unidad saldrá de esa lista en caso de que estuviera. Luego, otra función comprueba si esa lista contiene una unidad y, de ser el caso, la asigna a la variable Unit “unitOnMouse”. Aunque el proceso podría simplificarse (ya que no debería ser posible tener el puntero sobre varias unidades al mismo tiempo), se ha establecido así para *debugar* y asegurarse de todo está en orden, incluso en casos “imposibles” donde, por algún motivo, el área de una unidad opaca el área de otra.

Después, se desactivan todos los efectos visuales de selección que pudieran tener el resto de unidades, mientras que se activan los de unitOnMouse. Finalmente, se activa un *game object* de UI y muestra la información de la unidad. Además, otra función se mantiene revisando si unitOnMouse existe y al mismo tiempo el usuario hace clic. En caso afirmativo, la unidad se añade a la variable “selectedUnit”, lo que comporta que la unidad active su estado de “info selection” y se activen botones con las opciones disponibles, o, en otros casos, se seleccione a la unidad como un *target* de, por ejemplo, una carta en uso.

Todos estos efectos sobre las unidades se manejan a través de *flags*, por lo que la unidad puede tener activos varios efectos a la vez:

```
public enum UnitState
{
    None = 0,
    Soft = 1 << 0,
    Info = 1 << 1,
    Target = 1 << 2,
    Using = 1 << 3,
    Defending = 1 << 4,
    Meditation = 1 << 5
}
```

Estas flags son especialmente útiles para determinar los efectos visuales que se deben ver en la unidad sobre el tablero. Por cada *flag* activa, la unidad mostrará un borde de un color distintivo de cada estado. Además, cuanto mayor sea el número de estados activos, el tamaño (escalado) de estos se adaptará para que no se superpongan ni mezclen unos con otros, al mismo tiempo que se vigila que los efectos visuales más grandes no excedan un mayor espacio que la casilla donde se encuentra la unidad. Esto implica, no solamente aumentar el tamaño de los efectos cada vez más, sino también disminuir el tamaño de la propia unidad o de otros efectos, para mantener un equilibrio. Además, mientras la unidad está en modo “soft selection”, se aumenta más su tamaño y se posiciona a esta en una capa de renderizado superior al resto, para que se superponga a otras unidades en caso de sus *sprites* choquen ligeramente.

El tamaño de la interfaz de información de la unidad que se muestra es variable, ya que depende del resto de elementos visibles que comparten el espacio con este. Para lograr esto, se ha hecho uso del componente de UI “Vertical Layout Group”. El espacio dedicado a mostrar esa interfaz se ha definido como un rectángulo. Dentro de este, se han añadido individualmente los diferentes elementos que componen la información (base, imagen, icono de vida, etc.) y se han establecido los “anchor points” de cada uno para que mantengan su escala y relación cuando la base cambia de tamaño.

5.5.7. Cartas

Creación

Las cartas siguen un proceso de creación parecido a las unidades. Primero se llama a un RPC que genera una clase derivada CardData (esta depende del tipo de carta generada), y luego cada usuario instancia el prefab de ese tipo de carta en la escena (el cual tiene un *script* llamado “Card”), en su mano. La clase CardData contiene toda la posible información de la carta, desde la que es común en todos los tipos (como un nombre o descripción) a información específica del tipo de carta (como lo que hacen al

usarse o los propios valores específicos de esa carta). En muchas ocasiones, estos valores se determinan aleatoriamente en los constructores, como el valor de ataque o defensa, el tipo de combate que otorgarán a una unidad los índices de casillas afectadas.

Al igual que en las unidades, las instancias de CardData se almacenan en un diccionario del equipo (Team) correspondiente. Cada CardData tiene un número identificador "cardID". Los *scripts* Card de las cartas guardan este como "referenceCardId" al ser generados, para mantenerse comunicados con su CardData. Gracias a esto pueden, entre otras cosas, obtener la información que se ha mencionado anteriormente, como el nombre y descripción de la carta, y mostrarlo en su interfaz.

Tipos

Hay tres tipos de cartas principales: estadística, rango y especial. Cada uno de estos tiene varios subtipos. Para el primero: ataque y defensa. Para el segundo: invocación, movimiento, ataque y defensa. Para el tercero: activa y pasiva. Cada uno de estos subtipos tiene una plantilla o *template* (derivada de una clase padre "CardTemplate"), la cual se trata de un ScriptableObject que contiene cierta información referente a ese tipo de carta, pero no valores específicos de esa carta en concreto (los valores específicos de cada carta se encuentran en la clase CardData, como se ha mencionado).

Cada *template* tiene, entre otras cosas, un id distinto. Este se guarda en la variable "templateId" de CardData, para mantenerlos relacionados. Además, todos los *template* se añaden a una lista desde la que se puede acceder a ellos en cualquier momento. Otra información útil para identificar los templates es su variable cardType, la cual es un enum de tipo CardType con las siguientes opciones:

```
public enum CardType
{
    AttackStat,
    DefenseStat,

    InvocationRange,
    MovementRange,
    AttackRange,
    DefenseRange,

    ActiveSpecial,
    PassiveSpecial
}
```

Como se ha mencionado, cada subtipo de carta tiene un *template* con su información. Sin embargo, cabe destacar que en el caso de las cartas especiales, estas tienen un *template* por cada carta específica posible, donde se indica su nombre específico, descripción e imagen. Además, también tiene una variable enum con todas las cartas especiales específicas como opciones, el cual identifica la carta que es. Este enum se llama “specialCards”.

Efectos

Todos los *templates* tienen una variable “cardEffectScript” de tipo “BaseEffect”. La clase BaseEffect es una clase derivada de ScriptableObject, lo que permite ser adjuntada en la variable anteriormente mencionada. Esta clase tiene varios métodos relacionados con el uso de cartas en la partida. Al igual que cada tipo de carta usa un *template* de una clase hija de CardTemplate, cada tipo de carta también tiene una clase hija de BaseEffect. Así pues, haciendo varios *overrides* de funciones en cada caso, cada tipo de carta realiza unos efectos u otros al usarse. Aunque esto varía dependiendo del tipo de carta, todos siguen la misma estructura base:

1. Se obtiene el template de la carta que se quiere usar
2. Se obtiene el *script* de efecto asociado a ese template
3. Se añade la función principal de ese *script* a un evento de Unity, el cual se activa en cada *frame*
4. A partir de entonces, en cada llamada a esa función, se comprobará el estado actual del uso de la carta y se activarán unas funciones de acuerdo a este. El estado del uso de la carta se define según una variable enum con la siguiente estructura:

```
public enum CardEffectState
{
    None,
    StartUsing,
    Preparing,
    WaitingConfirmation,
    DoingEffect,
    EffectDone
}
```

Si todo eso se cumple, la carta hará su efecto, el cual está definido en una función *override* en cada tipo de carta. En general, este efecto consiste en hacer llamadas RPC que cambien atributos de alguna unidad, como lo es su ataque o defensa o rangos de acción.

En el caso de las cartas especiales, esto es un poco distinto; en vez de hacer algún cambio de atributos en la carta directamente, se le añade el id de su *template* a una lista de su UnitData. A partir de entonces, ya se puede identificar en todo momento las cartas especiales equipadas en una unidad y el usuario puede hacer uso de sus efectos. Para ello, se crearán unos *scripts* Monobehavior que ejecutan su código en Update() y se adjuntarán como componentes a un objeto llamado “CardEffectsManager”. Estos *scripts* se determinan mediante un diccionario, el cual relaciona valores del enum specialCards (los cuales, como se ha mencionado anteriormente, definen la carta especial y se encuentran establecidos en cada *template* de esa carta) con el componente a añadir (el tipo de clase).

Cada componente realiza efectos distintos en su Update(). En el caso de las cartas pasivas, estos efectos se ejecutan en todo momento, mientras que en el caso de las cartas activas, estos efectos se activan solamente cuando el usuario hace clic en el nombre de la carta, el cual es mostrado en la interfaz de información de la unidad.

Además de funciones, los *scripts* también contienen otra información útil como el id de la unidad en la que ese efecto está “equipado”, lista de unidades marcadas como target, rango de casillas en las que tiene efecto la carta, número de usos de la carta, etc.

Imagen de rangos

Las cartas de estadística muestran como texto su valor de estadística concreto y muestran el tipo de combate con una imagen. Las cartas especiales muestran una imagen asociada específicamente a esa carta, en su *template*. En cambio, las cartas de rango deben mostrar una información mucho más compleja, la cual debe ser fácilmente identificable y distinguible: las casillas afectadas por esa carta, respecto la posición de la unidad que use la acción.

Para representar esto, se ha recurrido, en primer lugar, a la misma técnica empleada para renderizar el tablero de juego. Con esto se puede representar de una manera muy fácil un pequeño tablero en cada carta, dentro del cual se añadirán imágenes de una unidad y las casillas de rango. Para destacar estos últimos elementos por encima de todo, el tablero usará colores apagados, como tonalidades de marrón.

Además, el tamaño del tablero se adaptará para ser lo más claro posible. Esto quiere decir que no se dejarán casillas vacías en los bordes que no aporten información o no

sean necesarias, pudiendo aprovechar ese espacio para aumentar el tamaño del resto de elementos.

Para definir el tamaño de estos tableros, se comprueba cuál es el mayor valor (absoluto) de las coordenadas de las casillas de rango a mostrar y se suma 1. Esto es así ya que la unidad debe mostrarse también (siempre en el "(0,0)"), por lo que si se muestra la unidad y, por ejemplo, la casilla (0,1) de delante de ella, se necesita un tablero de 2x2. Después, al igual que con el tablero de juego, se determina el tamaño de los elementos en base al tamaño de las casillas.

Luego, tanto en el eje X como en el Y, se obtienen los índices de las casillas de rango con el menor y el mayor valor. En caso de que el menor valor sea negativo, se añade ese valor como absoluto (o, lo que es lo mismo, la diferencia entre 0 y ese valor) a las variables int "xToRemoveNegative" e "yToRemoveNegative". Esta variable se sumará más adelante a los índices de la unidad y de las casillas de rango para determinar su posición en el nuevo tablero. Esto es porque los índices de las casillas de este tablero no pueden ser negativos, así que se adaptan los índices de los elementos incluidos en este.

Además, también se obtiene la diferencia entre el número de casillas por eje del tablero y el nuevo índice de la casilla con rango con el mayor valor. De este manera se obtiene el número de casillas que quedan vacías. Este número se divide entre 2 y el resultado (sin decimales) se guardará en las variables "xToCenter" e "yToCenter", las cuales nuevamente serán valores que se sumarán a los índices de los elementos del tablero. Esta vez, esto servirá para centrar los elementos en aquellos casos en los que un eje no quede tan aprovechado como el otro y, por lo tanto, este último tiene un espacio de casillas vacías en un lado que puede aprovecharse para centrar la información.

Una vez se ha determinado el tamaño que deben tener los elementos y su índice en el nuevo tablero, se sitúan *game objects* con sprites de esos elementos en la posición que les corresponde.

5.6. Creación del arte del juego

Todo el arte del juego es original (o usa recursos básicos del propio Unity). Elementos como iconos o interfaces de información de las unidades o cartas han sido creados con

Figma. Por otro lado, elementos como la madera del fondo o del borde del tablero o las imágenes de las unidades y las cartas especiales han sido generados a través de IA.

Las texturas de madera han sido generadas usando el modelo general de la versión 1.5 de Stable Diffusion. Las imágenes de las unidades y cartas, en cambio, han hecho uso de la versión 4.0 del modelo “Arthemy Comics”. (*Arthemy Comics - v4.0 BakedVAE | Stable Diffusion Checkpoint | Civitai*, 2024)

Las imágenes de las cartas especiales han sido creadas a partir de *prompts* específicos. Por otro lado, todas las imágenes de las unidades han seguido una estructura de *prompts* muy parecida que da consistencia al conjunto y agilidad en el proceso de creación, pero con algunas palabras variables que le dan diversidad a cada personaje representado.

Para llevar esto a cabo, se ha creado un programa simple en C++ usando Visual Studio, el cual es un generador de *prompts* que genera varios *prompts* y los escribe en un documento “.txt”, cada uno en una línea. Cada uno de esos *prompts* tiene ciertas palabras o sintagmas clave que se han escogido de manera aleatoria entre un listado, de manera individual.

Una vez se obtiene el document de texto, se puede usar su *path* en la interfaz de ComfyUI para que el programa obtenga el prompt de la línea seleccionada. Esto se hace, concretamente, a través del nodo “Text Load Line From File”, el cual pertenece a la extension WAS Node Suite. Con cada orden de generación que se haga, el índice de línea seleccionada avanzará.

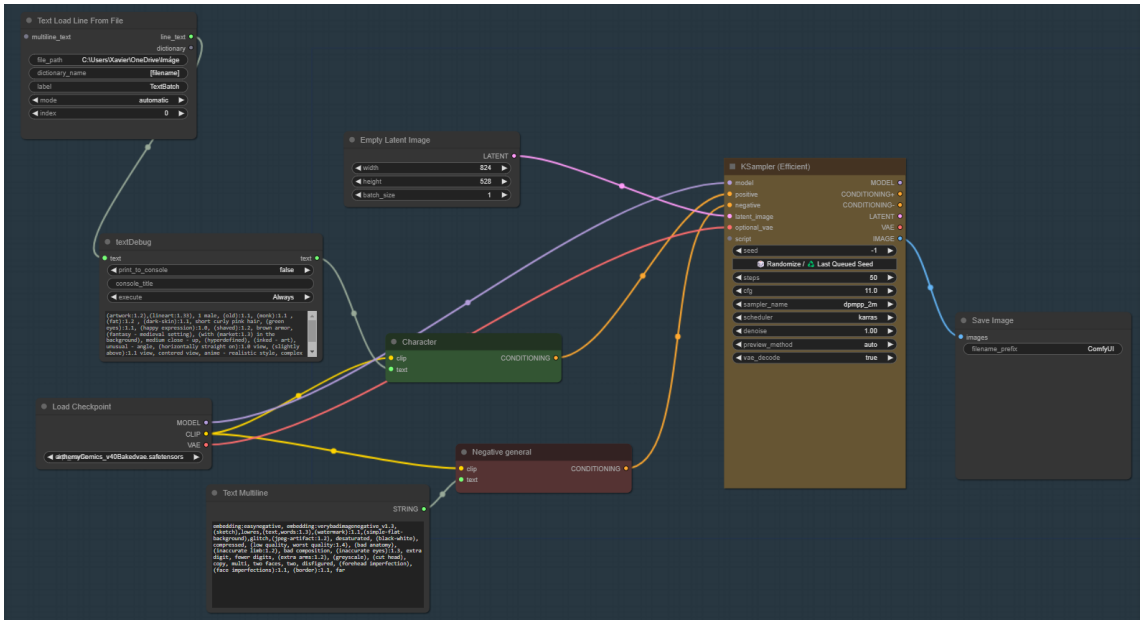

Figura 27. Workflow de ComfyUI

Este es el workflow utilizado para generar las imágenes de las unidades del juego.

5.7. Añadido de funcionalidades y detalles

Una funcionalidad útil que se ha añadido al prototipo son los *tooltips*. Estos son pequeños recuadros con texto que aparecen en pantalla cuando el jugador mantiene el cursor encima de un elemento del juego. Este recuadro aparece al lado del elemento y su texto le da información al usuario sobre este. Además, el tamaño del recuadro se adapta a la cantidad de texto. Esto ayudará a los usuarios en el testeo del prototipo.

El añadido de esta funcionalidad se basa en dos *scripts*: “Tooltip” y “TooltipTrigger”. El primero se añade como componente al propio elemento de UI que se muestra en pantalla. Este tiene sobre todo dos funciones; una para mostrar el objeto (en cierta posición que se calcula según los parámetros dados) y otra para esconderlo. TooltipTrigger es un componente que se añade en aquellos objetos que deben activar el tooltip. El componente detecta cuándo el puntero del ratón “entra” en este objeto y cuando sale. Entonces, llama a la función de activar o desactivar el tooltip, respectivamente.

Al llamar a la primera función, se envían ciertos parámetros. Uno de ellos es el propio texto que se quiere mostrar en el tooltip. Otro es la posición del propio objeto sobre el que se indica la información (portador del TooltipTrigger), que indica una posición básica sobre la que mostrar el tooltip. También se envía un bool que indica si el tooltip se

mostrará a la izquierda o la derecha de esa posición, y un offset personalizable que añade cierta distancia sobre esa posición.

Además, se usa una coroutine antes de hacer la llamada para activar el tooltip. Esta espera un poco de tiempo antes de hacerlo, para que esta herramienta no moleste a los jugadores que solamente pasan el ratón por encima de un elemento, pero no se detienen en este porque no están interesados en ver su información.

Por último, también se han tenido en cuenta los bordes de la pantalla para asegurarse de que el tooltip no los sobrepasa.

6. Validación del proyecto

El prototipo ha sido testeado por varias personas, y se ha recogido *feedback* al respecto de distintas maneras. Además, aquellos problemas más destacables que dificultaban una buena jugabilidad han sido corregidos (además de *bugs*) después de ser detectados, por lo que durante el transcurso del periodo de *playtesting*, se han ido realizando iteraciones en el juego para mejorar la experiencia a futuro.

6.1. Método

No se ha usado un único método de *testing*, sino una combinación de varios o método mixto. Esto es debido a que se han querido aprovechar ciertos aspectos ventajosos de ambos tipos de metodologías, tanto cualitativas como cuantitativas. Los métodos cualitativos aportan un gran valor a las conclusiones, ya que se tiene un trato mucho más cercano y personalizado con el participante y, por lo general, se le entiende mejor. Los métodos cuantitativos usados, por otra parte, nos permiten tener un punto de vista más “frío” y “medible”, y en muchas ocasiones más honesto.

Respecto a esto, es importante destacar que, en cualquier proceso llevado a cabo, en todo momento se le ha hecho saber al usuario que sus opiniones deben ser lo más honestas posibles.

Por lo general, el proceso de *playtesting* que se ha seguido inicia con una explicación de las normas de juego, luego el usuario prueba el juego mientras se observan sus comportamientos. Al terminar, el usuario rellena un formulario y se le entrevista para conocer su opinión.

6.1.1. Explicación de las normas

Se le explican las normas al participante. Esto es acompañado de una imagen con la información más relevante, la cual el usuario puede usar en todo momento como guía y consultarla durante el juego.

6.1.2. Prueba de usabilidad

Este es un proceso cualitativo. El participante juega al prototipo y se le monitorea en todo momento mientras lo hace. Se escuchan comentarios u opiniones que este pueda tener en medio de las partidas y se observan sus comportamientos. Se juega más de

una partida, ya que la primera sirve en gran parte para que usuario entienda las bases del juego y aprenda el funcionamiento práctico de la partida.

6.1.3. Formulario

Este es un proceso cuantitativo. El participante rellena un formulario anónimo y se obtiene información sobre él, su experiencia y sus opiniones, tanto en algunos temas más concretos como en un nivel más general. Esto permite comparar las respuestas de distintos participantes de una manera fácil.

6.1.4. Entrevista

Este es, de nuevo, un proceso cualitativo. Se habla directamente con el participante para saber sus sensaciones y pensamientos en profundidad. Aunque no se debe interpretar la opinión de un solo individuo como una verdad absoluta, estas opiniones, por lo general, tienen ciertos fundamentos y deben tenerse en consideración, ya que pueden ser de mucha utilidad y/o ser compartidos por más personas.

6.1.5. Focus groups

En algunas ocasiones se han llevado a cabo reuniones de varios participantes (*focus groups*) y estos han puesto en común algunos puntos y pensamientos. Esto es un proceso cualitativo que ha sido útil especialmente para determinar aquellas opiniones comunes.

6.2. Resultados

6.2.1. Ajustes realizados

Como se ha mencionado anteriormente, se han llevado a cabo algunos ajustes en el prototipo durante el playtesting para corregir problemas importantes detectados. Estos se explican a continuación:

- Ajuste de precios: Los costes de distintos elementos del juego se han ajustado. En su mayoría, se han reducido para ofrecer más simplicidad y dinamismo. Robar una carta ahora cuesta 2 de oro independientemente del tipo de carta (antes las de estadística y especiales costaban 3). Comprar un hueco de rango en una unidad (es decir, aumentar

su nivel de acción) ahora cuesta 5 de oro independientemente del nivel (antes el coste aumentaba por nivel).

- Rotación de unidades: Se ha añadido la opción de rotar las unidades 90 grados en sentido horario. Esto originalmente era un efecto de una carta especial, pero ese efecto era extremadamente útil y desbalanceado. Por otro lado, ofrecía una mecánica muy interesante y que combinaba bien con las demás, por lo que, en vez de simplemente eliminarlo, se planteó su incorporación al juego como otra acción más que podía usar cualquier unidad. Sin embargo, esto se limitó a un uso por turno.

-Rangos: Las cartas de rango originalmente ofrecían solo una casilla. Esto dificultaba el juego hasta el punto de ser prácticamente injugable, puesto que era muy complicado hacer coincidir las casillas de ataque o defensa con unidades a las que afectar, además combinarlas bien con las casillas de movimiento. Por eso, se amplió el número de casillas a 4 por carta. Para simplificar la jugabilidad, las casillas són adyacentes, una al lado de la otra, por lo que se forman “areas” donde interactuar. En un principio estas areas eran cuadradas, pero posteriormente adquirieron una forma aleatoria para ofrecer más variedad y combinaciones más interesantes entre diferentes rangos. Además, la distancia entre esos rangos y la unidad usuaria se redujo una casilla para simplificar los cálculos y para que las unidades deban acercarse más entre sí.

-Invocaciones: Las invocaciones eran un recurso muy fácil de usar y que podia alargar la partida prácticamente hasta la infinidad en algunas ocasiones, además de anular el progreso o continuidad de la partida. Es por eso que se ha tomado la decisión de eliminarlas del juego como acciones. Así pues, las invocaciones ahora solo se realizan al inicio de la partida para establecer las unidades de ambos jugadores en su primer turno, y en vez de una sola se invocan 3, número que no aumentará durante la partida.

-Acciones: Que una unidad llevase a cabo múltiples acciones durante un mismo turno es algo que, si bien ofrece dinamismo y complejidad al juego y le permite ser creativo en sus estrategias al jugador, podia resultar injusto para el adversario que “recibía” estas acciones, sin que este tuviera muchas oportunidades de prepararse para ello o afrontarlo. Aunque originalmente se había planteado el oro del jugador como factor limitante de estas acciones, este no resultó ser lo suficientemente limitante, además de ser algo difícil de controlar para el que se tiene que preparar para recibir estas acciones. Debido a esto, se ha obtado por limitar el uso de estas acciones problemáticas de una manera mucho más directa. Ahora, cada unidad solo puede atacar y moverse una sola

vez por turno. Si bien es cierto que esto hacer perder un poco el dinamismo de la partida, también hay que considerar que se ha agregado una nueva acción: la rotación, por lo que los jugadores siguen disponiendo de formas creativas de idear sus estrategias mientras que la jugabilidad se ha vuelto mucho más justa.

-Vida: Aumentar la vida con dinero era un recurso que inicialmente no se usaba mucho por destinar el oro a otros elementos. Primero se redujo su precio a 2 de oro para cualquier nivel (en vez de un precio que incrementaba), pero aumentar la vida seguía sin ser una mecánica que estuviera muy en harmonía con el resto del juego. Así pues, ahora la estadística de vida no se incrementa con oro, sino que tiene una carta de estadística propia para aplicar a las unidades, al igual que el ataque y la defensa.

-Defensa: La defensa es un elemento que no es muy usado, especialmente cuando el jugador no tiene mucha experiencia y se centra en otros aspectos del juego. Se ha incrementado su utilidad añadiendo que una unidad defendiendose activamente usa su defensa para protegerse independientemente de si el enemigo se encuentra en su rango de defensa.

6.2.2. Feedback

Una vez se ha ajustado el prototipo a una versión mucho más estable y jugable, se han llevado a cabo la mayoría de los *playtestings* mencionados anteriormente. A continuación se muestran algunas opiniones destacables:

- Unidades más resistentes: Se puede considerar aumentar la vida base de las unidades a 10 y que se pueda aumentar hasta unos 20. Esto las haría más resistentes e impediría en gran medida que murieran de un solo ataque, situación que es bastante frecuente. Alternativamente también se podría disminuir el ataque a una estadística máxima de 5, ya que actualmente una unidad con más de ese ataque que tiene una ventaja de tipo de combate elimina de forma casi segura a cualquier enemigo independientemente de su vida (si no se está defendiendo).

-Unidades de diferentes tipos al invocar: Las tres unidades invocadas al inicio podrían ser una de cada tipo de combate. Esto puede reforzar las estrategias tempranas.

-Escoger la dirección de rotación: Es una de las sugerencias más populares. Esto haría más fácil llevar a cabo algunas estrategias y, por consiguiente, agilizaría el juego.

-Agilizar el inicio de la partida: El inicio puede ser un poco tedioso para los jugadores, ya que hay mucha más preparación que acción.

-Disminuir la importancia de la economía: La economía supone para muchos jugadores un obstáculo frustrante. Varios de ellos piensan que se podría prescindir mucho más de su uso. Esto cobra especial sentido teniendo en cuenta que uno de los principales motivos de esta era regular el número de acciones que realizan las unidades por turno, pero tras los ajustes llevados a cabo, esto finalmente se hace a través de un límite de usos por unidad.

-Regular la aleatoriedad: Resulta muy frustrante para los jugadores que las cartas que roban sean, en muchas ocasiones, no del tipo o valor que ellos querían, en especial si es frecuente el robo de estas cartas que ellos no consideran útiles en su situación. Esto es, por ejemplo, buscar una carta de estadística de defensa y robar 5 cartas de las cuales ninguna cumple con eso, o robar dos veces seguidas una misma carta especial. El poder vender 2 cartas y obtener dinero para comprar otra es algo que se pensó para mitigar este problema, pero puede no ser suficiente. Para solucionar esto, se podría aplicar algún algoritmo o técnica que tenga en cuenta la situación actual del jugador y regule el contenido aleatorio nuevo que se genera, buscando que el jugador disponga de cartas variadas y no similares o con elementos repetidos. Es decir, habría que convertir este tipo de aleatoriedad en una uniforme, regulada.

Finalmente, otro aspecto a comentar es la paridad entre las condiciones de los dos jugadores, ya que creo que se ha llegado a un buen equilibrio. En primer lugar, el primer jugador tiene la ligera ventaja de empezar antes y, por lo tanto, ir un turno “por delante” del segundo. Por otro lado, el segundo jugador sabe donde ha puesto las unidades el primero, por lo que puede posicionar las suyas ajustandose ligeramente a eso (aunque limitado en su zona).

7.Conclusiones

Pese a tratarse de un prototipo bastante básico a nivel de pulido, Karcha transmite muy bien la idea que se buscaba. Es una base bastante sólida sobre la que se puede construir. Hay varios ajustes y detalles que se pueden aplicar al sistema de juego, o incluso se pueden alterar algunas mecánicas, pero tiene una identidad y base lo suficientemente compacta y funcional como para determinar que es posible y factible crear un juego con un gran componente de aleatoriedad como lo es un sistema de cartas pero que aun así mantenga un sistema de juego estratégico y competitivo similar al del ajedrez.

El juego ha sido probado por varios tipos de jugadores, y tanto aquellos que se enfocan más en la competitividad como aquellos que prefieren disfrutar de otras maneras han determinado que les ha parecido divertido jugar y han querido jugar a más partidas de las estrictamente necesarias, por voluntad propia.

El elemento de aleatoriedad contribuye enormemente a esto, ya que cada partida se desarrolla de una manera totalmente distinta y no saber qué cartas te tocarán en tu próxima partida y qué movimientos podrás realizar fomenta mucho el querer iniciar una nueva partida y descubrirlo.

Del mismo modo, el jugador debe tomar una gran cantidad de decisiones importantes, desde cómo y dónde usar las cartas hasta qué acciones realizar en el tablero con sus piezas, que es lo que realmente determina el desarrollo y resultado de la partida. Es decir, la aleatoriedad no opaca la estrategia del jugador, y por lo tanto cuando un jugador elimina una pieza del rival, el primero se siente muy satisfecho con su habilidad y el segundo se lamenta de no haber visto con antelación la jugada, pero en ningún caso estas emociones recaen sobre la aleatoriedad (lo que se traduciría en que el primero apenas sentiría satisfacción y el segundo lo sentiría como una injusticia).

Como resultado, los objetivos principales de este proyecto se han cumplido satisfactoriamente.

7.1. Líneas de futuro

En caso de querer desarrollar más en profundidad el proyecto y acercarlo más a lo que sería una version final y completa de un juego, hay varias acciones que se pueden llevar a cabo, algunas de ellas mencionadas por los *testers* del prototipo:

- Regular la aleatoriedad para dificultar que ocurran casos extremadamente negativos o positivos como robar varias cartas seguidas del mismo tipo o varias cartas de estadística con los valores máximos, respectivamente.
- Añadir audio al juego, tanto música como efectos de sonido.
- Pulir aspectos gráficos y especialmente *feedback* (visual y sonoro) al realizar acciones o suceder eventos como un cambio de turno.
- Plantear si las interfaces podrían mejorar su accesibilidad y si la manera en la que el jugador interactúa con el juego podría ser más práctica o cómoda para él.
- Realizar ajustes de balance, especialmente del sistema de economía del juego.
- Agregar funcionalidades como un chat, modo espectador y registro de acciones.
- Añadir más cartas especiales únicas.

Como bien se ha mencionado anteriormente, en este punto Karcha es un prototipo que ha confirmado la viabilidad práctica del proyecto. A partir de aquí, en caso de querer convertirlo en un producto final, se debería de trabajar en muchos aspectos, pero la base está construida.

8. Bibliografía

A Study in Transparency: How Board Games Matter. (s. f.). Recuperado 2 de marzo de 2024, de https://www.gdcvault.com/play/1020408/A-Study-in-Transparency-How

Aguilla, C. (2022, agosto 21). *Generación procedural en los videojuegos: Cuando las matemáticas facilitan el trabajo creativo*. Xataka México. https://www.xataka.com.mx/videojuegos/generacion-procedural-videojuegos-cuando-matematicas-facilitan-trabajo-creativo

Amaranth. (s. f.). Google Fonts. Recuperado 3 de mayo de 2024, de https://fonts.google.com/specimen/Amaranth

Analysis paralysis. (2024). En *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Analysis_paralysis&oldid=1203310607

Arthemy Comics—V4.0 BakedVAE | Stable Diffusion Checkpoint | Civitai. (2024, marzo 6). https://civitai.com/models/54073/arthemy-comics

Board Game Design Day: Balancing Mechanics for Your Card Game's Unique Power Curve—YouTube. (s. f.). Recuperado 2 de marzo de 2024, de https://www.youtube.com/watch?v=ul1MSQ8aW00

Build software better, together. (s. f.). GitHub. Recuperado 8 de marzo de 2024, de https://github.com

Caillois, R. (1958). *Les jeux et les hommes*.

Caillois, R., & Barash, M. (2001). *Man, play, and games* (1. Illinois paperback). Univ. of Illinois Press.

Chessboard. (2024). En *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Chessboard&oldid=1208815053

Chess.com—Play Chess Online—Free Games. (s. f.). Chess.Com. Recuperado 4 de marzo de 2024, de https://www.chess.com/

Civitai: The Home of Open-Source Generative AI. (s. f.). Recuperado 8 de marzo de 2024, de https://civitai.com/

comfyanonymous. (2024). *Comfyanonymous/ComfyUI* [Python]. https://github.com/comfyanonymous/ComfyUI (Obra original publicada en 2023)

de 3DJuegos, E. equipo. (2021, abril 6). *Juegos con aleatoriedad: ¿Es mejor que nos limiten para fomentar la rejugabilidad?* 3DJuegos. https://www.3djuegos.com/pc/noticias/juegos-con-aleatoriedad-es-mejor-que-nos-limiten-para-fomentar-la-rejugabilidad-210406-2262

Desarrollo en cascada. (2024). En *Wikipedia, la enciclopedia libre*. https://es.wikipedia.org/w/index.php?title=Desarrollo_en_cascada&oldid=158529069

Dice chess. (2023). En *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Dice_chess&oldid=1165076878

Dissection of Randomness in Games. (s. f.). Recuperado 2 de marzo de 2024, de https://www.gamedeveloper.com/design/dissection-of-randomness-in-games

Donlan, C. (2019, enero 30). *Slay the Spire review—A gorgeous blend of dungeon-crawler and card-battler*. https://www.eurogamer.net/slay-the-spire-review-a-gorgeous-blend-of-dungeon-crawler-and-card-battler

Eggplant: The Secret Lives of Games: 12: Into the Breach with Justin Ma. (s. f.). Recuperado 4 de marzo de 2024, de https://eggplant.show/12-into-the-breach-with-justin-ma

Emil_lab. (2024, marzo 2). *Qué es juego (2)* | *El Lab de Emil-lab*. https://emil-lab.eu/que-es-jugar-2

Figma: The Collaborative Interface Design Tool. (s. f.). Figma. Recuperado 8 de mayo de 2024, de https://www.figma.com/

FirstGearGames. (2024). *FirstGearGames/FishySteamworks* [C#]. https://github.com/FirstGearGames/FishySteamworks (Obra original publicada en 2021)

Fischer random chess. (2024). En *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Fischer_random_chess&oldid=1211252085

Free Online Gantt Chart Software. (s. f.). Recuperado 7 de marzo de 2024, de https://www.onlinegantt.com

GitHub Desktop. (s. f.). GitHub Desktop. Recuperado 8 de marzo de 2024, de https://desktop.github.com/

Goodman, J., & Irwin, J. (2006). Special random numbers: Beyond the illusion of control. *Organizational Behavior and Human Decision Processes*, *99*, 161-174. https://doi.org/10.1016/j.obhdp.2005.08.004

Hearthstone. (2024). En *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Hearthstone&oldid=1206790222

Hoeppner, E. (s. f.). *Plan Disruption*. https://ethanhoeppner.github.io/gamedesign/plan-disruption.html

How to play Dice Chess—YouTube. (s. f.). Recuperado 2 de marzo de 2024, de https://www.youtube.com/watch?v=UDW5qaW7lqo

Inicio. (s. f.). Canva. Recuperado 6 de marzo de 2024, de https://www.canva.com/

Introduction. (s. f.). Recuperado 8 de marzo de 2024, de https://fish-networking.gitbook.io/docs/

Jed Herne (Director). (2022, septiembre 7). *I made a board game! Introducing: Not Chess*. https://www.youtube.com/watch?v=20sA_pcoWRc

Karar2k's Why Play This (Director). (2022, octubre 13). *Not Chess—Game trailer*. https://www.youtube.com/watch?v=BdKtky-KgwE

Keith Burgun Games (Director). (2014, diciembre 2). *3 Minute Game Design: Episode 4 - Elegance and Depth*. https://www.youtube.com/watch?v=IMsFHCdqjTk

Knightmare Chess. (2023). En *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Knightmare_Chess&oldid=1190397871

Malaby, T. M. (2007). Beyond Play: A New Approach to Games. *Games and Culture*, *2*(2), 95-113. https://doi.org/10.1177/1555412007299434

No Stress Chess. (s. f.). BoardGameGeek. Recuperado 2 de marzo de 2024, de https://boardgamegeek.com/boardgame/19918/no-stress-chess

No Stress Chess: Amazon.co.uk: Toys & Games. (s. f.). Recuperado 2 de marzo de 2024, de https://www.amazon.co.uk/Winning-Moves-1091-Stress-Chess/dp/B0007Q1IO4

No Stress Chess Game Just $12.40 on Amazon (Reg. $20) | Over 5,900 5-Star Ratings. (2023, agosto 14). *Hip2Save*. https://hip2save.com/deals/no-stress-chess-game/

NOT CHESS: A party card game expansion for chess (Canceled). (2023, abril 1). Kickstarter. https://www.kickstarter.com/projects/goodlarkgames/not-chess-a-party-card-game-expansion-for-chess

Nugie Romantic Font | *dafont.com*. (s. f.). Recuperado 5 de mayo de 2024, de https://www.dafont.com/nugie-romantic.font

Parker, J. (s. f.). *Hearthstone for iOS review: A strategy card game anyone can enjoy*. CNET. Recuperado 4 de marzo de 2024, de https://www.cnet.com/reviews/hearthstone-ios-review/

Plataforma de desarrollo en tiempo real de Unity | *Motor de 3D, 2D, VR y AR*. (s. f.). Unity. Recuperado 8 de marzo de 2024, de https://unity.com/

Randomness and Game Design. (2014, octubre 14). *KEITH BURGUN GAMES*. https://keithburgun.net/randomness-and-game-design/

Randomness in Games: A Long-form Analysis—YouTube. (s. f.). Recuperado 2 de marzo de 2024, de https://www.youtube.com/watch?v=QNs8aB0huoc

Sandboxie-Plus | *Open Source sandbox-based isolation software*. (s. f.). Recuperado 8 de marzo de 2024, de https://sandboxie-plus.com/

Slay the Spire. (2024). En *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Slay_the_Spire&oldid=1210283057

Stability AI Image Models. (s. f.). Stability AI. Recuperado 8 de marzo de 2024, de https://stability.ai/stable-image

Steam, The Ultimate Online Game Platform. (s. f.). Recuperado 8 de marzo de 2024, de https://store.steampowered.com/about/

Steamworks.NET - Steamworks.NET. (s. f.). Recuperado 8 de marzo de 2024, de https://steamworks.github.io/

Sueldo: Junior Game Developer en Barcelona, Spain 2024. (s. f.). Glassdoor. Recuperado 6 de marzo de 2024, de https://www.glassdoor.es/Sueldos/junior-game-developer-sueldo-SRCH_IM1015_KO0,21.htm

Thanks for letting me check this out @NOT CHESS! #chess #boardgame #k... | *TikTok*. (s. f.). Recuperado 4 de marzo de 2024, de https://www.tiktok.com/@boredboardgames/video/7150674116566306094

The Delta of Randomness—Can You Balance for RNG? - Extra Credits—YouTube. (s. f.). Recuperado 2 de marzo de 2024, de https://www.youtube.com/watch?v=0V5eq4IQ6Go

The Two Types of Random in Game Design—YouTube. (s. f.). Recuperado 2 de marzo de 2024, de https://www.youtube.com/watch?v=dwI5b-wRLic

thebeanone. (2023, noviembre 30). *Why would my chess set come with two die?* [Reddit Post]. r/chess. www.reddit.com/r/chess/comments/187b01l/why_would_my_chess_set_come_with_two_die/

Three types of bad randomness, and one good one. (2018, mayo 22). *KEITH BURGUN GAMES*. https://keithburgun.net/three-types-of-bad-randomness-and-one-good-one/

Triple S Games (Director). (2021, noviembre 30). *How to play No Stress Chess*. https://www.youtube.com/watch?v=lcqApfXfzv4

Triple S Games (Director). (2023a, agosto 14). *How to play Uno Chess*. https://www.youtube.com/watch?v=vzkKs_b5css

Triple S Games (Director). (2023b, septiembre 19). *How to play Knightmare Chess*. https://www.youtube.com/watch?v=z_xfnApKhMU

True Hit. (s. f.). *Serenes Forest*. Recuperado 4 de marzo de 2024, de https://serenesforest.net/general/true-hit/

Uncapped Look-Ahead and the Information Horizon. (2014, diciembre 30). *KEITH BURGUN GAMES*. https://keithburgun.net/uncapped-look-ahead-and-the-information-horizon/

«Uncertainty in Games» Greg Costikyan, Playdom—YouTube. (s. f.). Recuperado 2 de marzo de 2024, de https://www.youtube.com/watch?v=qXk96RK8qpo

Visual Studio 2022 Community Edition: Descargar la versión gratuita más reciente. (s. f.). Visual Studio. Recuperado 8 de marzo de 2024, de https://visualstudio.microsoft.com/es/vs/community/

Way Too Many Games Were Released On Steam In 2023. (2024, enero 2). Kotaku. https://kotaku.com/steam-pc-new-releases-valve-2023-1851133788

Zhang, Y., Monteiro, D., Liang, H.-N., Ma, J., & Baghaei, N. (2021). Effect of Input-output Randomness on Gameplay Satisfaction in Collectable Card Games. *2021 IEEE Conference on Games (CoG)*, 01-05. https://doi.org/10.1109/CoG52621.2021.9619020