Universidad Carlos III de Madrid

Ingeniería en Informática

Proyecto Fin de Carrera

MAGIC QUEST, DESARROLLO DE UN VIDEOJUEGO CON XNA

Autor: José Carlos Redondo Sánchez
Tutor: Daniel Borrajo Millán

Octubre de 2010

Agradecimientos

A José María Redondo y Catalina Sánchez, mis padres, que me han aportado una gran educación, guiándome y ayudándome en todo lo posible. Son los verdaderos responsables de que haya llegado hasta donde estoy.

A Iván, mi primo, que siempre me ha valorado y me ha ayudado con las ideas que le iba comentando a lo largo del desarrollo del juego, y ha probado y juzgado mi juego, aportándome siempre su punto de vista.

A Rosa, que siempre ha valorado todo lo que hacía, que me ha soportado muchos momentos mientras yo estaba desarrollando este PFC, porque sabía que se trataba de algo muy importante para mí. Gracias por todos esos momentos.

A Román y Juanjo, grandes compañeros y amigos, que me han dado ideas y me han asesorado a la hora de tomar muchas decisiones aportándome su conocimiento y experiencia.

Al resto de mis compañeros de trabajo, mención especial para Eva, que siempre me han animado e incluso gran parte de ellos han probado el juego y me han dado su opinión.

A Daniel Borrajo, mi tutor, que me ha dado la oportunidad de desarrollar este PFC, y siempre ha sabido escuchar mis opiniones y guiarme, aportándome ideas y sugerencias, orientándome hasta el resultado obtenido.

Índice de contenidos

AGRADECIMIENTOS III
ÍNDICE DE CONTENIDOS IV
ÍNDICE DE ILUSTRACIONES VI
ÍNDICE DE TABLAS VII
1 INTRODUCCIÓN 8
2 ESTADO DE LA CUESTIÓN 11
2.1 HISTORIA Y EVOLUCIÓN DE LOS VIDEOJUEGOS 11
2.2 HERRAMIENTA DE DESARROLLO: XNA 16
2.2.1 Introducción 16
2.2.2 Breve historia de XNA 18
2.2.3 Elección de XNA para el desarrollo del proyecto 20
2.2.4 Componentes a instalar para usar XNA y requisitos mínimos 22
2.2.5 Descripción de la plantilla inicial 23
2.3 VIDEOJUEGO PUZZLE QUEST 25
2.4 INTELIGENCIA ARTIFICIAL EN LOS VIDEOJUEGOS 28
2.4.1 Técnicas de Inteligencia Artificial 28
2.4.2 Técnica utilizada: árboles y reglas de decisión 32
2.4.3 Herramienta utilizada: Weka 33
3 OBJETIVOS 34
4 DESARROLLO 36
4.1 INTRODUCCIÓN 36
4.2 ANÁLISIS 36
4.2.1 Actores 36
4.2.2 Casos de uso 38
4.3 DISEÑO CONCEPTUAL 40
4.3.1 Arquitectura de la aplicación 40
4.3.2 Capas de la arquitectura 43
4.4 DISEÑO DETALLADO 45
4.4.1 Módulo de entrada 45
4.4.2 Módulo de salida 47
4.4.3 Módulo de componentes 49
4.4.4 Módulo de juego 56
4.4.5 Módulo de aplicación 66
4.4.6 Módulo de configuración 69
4.4.7 Módulo de log 71
4.4.8 Módulo de IA 76
4.5 JUGADOR AUTOMÁTICO 78
4.5.1 Definición de los logs 78
4.5.2 Selección de clasificadores 85
4.5.3 Experimentación con Weka 86
4.5.4 Implementación del jugador automático 98
4.5.5 Reglas de decisión obtenidas 100
5 RESULTADOS 102
6 CONCLUSIONES 103
7 LÍNEAS FUTURAS 105

8 GESTIÓN DE PROYECTO 106

8.1 PLANIFICACIÓN.....106
8.2 PRESUPUESTO 108
8.2.1 Costes de personal.....108
8.2.2 Costes de equipamiento.....108
8.2.3 Costes totales.....109

9 BIBLIOGRAFÍA 110

ANEXO I: COMPONENTES A INSTALAR PARA USAR XNA Y REQUISITOS MÍNIMOS 111

ANEXO II: MANUAL DE USUARIO 113

Índice de ilustraciones

ILUSTRACIÓN 1-1: PUZZLE QUEST ORIGINAL 9
ILUSTRACIÓN 2-1: LANZAMIENTO DE MISILES 12
ILUSTRACIÓN 2-2: TRES EN RAYA 12
ILUSTRACIÓN 2-3: SPACEWAR 12
ILUSTRACIÓN 2-4: PONG 12
ILUSTRACIÓN 2-5: MEGA DRIVE 14
ILUSTRACIÓN 2-6: SUPER NINTENDO 14
ILUSTRACIÓN 2-7: PLAYSTATION 15
ILUSTRACIÓN 2-8: NINTENDO 64 15
ILUSTRACIÓN 2-9: WII, PLAYSTATION 3, XBOX 360 16
ILUSTRACIÓN 2-10: COMPILACIÓN DE .NET 17
ILUSTRACIÓN 2-11: CAPAS DE XNA 18
ILUSTRACIÓN 2-12: PUZZLE QUEST - MAPA 25
ILUSTRACIÓN 2-13: TIPOS DE CASILLAS EN PUZZLE QUEST 26
ILUSTRACIÓN 2-14: COMBATE EN PUZZLE QUEST 27
ILUSTRACIÓN 2-15: ÁRBOL DE DECISIÓN PARA JUGAR AL TENIS 29
ILUSTRACIÓN 2-16: FASES DE LOS ALGORITMOS GENÉTICOS 30
ILUSTRACIÓN 2-17: ESTRUCTURA DE UNA RED DE NEURONAS 31
ILUSTRACIÓN 4-1: DIAGRAMA DE CASOS DE USO 38
ILUSTRACIÓN 4-2: DIAGRAMA DE COMPONENTES 41
ILUSTRACIÓN 4-3: CAPAS DE LA ARQUITECTURA 44
ILUSTRACIÓN 4-4: MÓDULO DE ENTRADA 45
ILUSTRACIÓN 4-5: MÓDULO DE SALIDA 47
ILUSTRACIÓN 4-6: MÓDULO DE COMPONENTES 49
ILUSTRACIÓN 4-7: MÓDULO DE JUEGO 56
ILUSTRACIÓN 4-8: MÓDULO DE APLICACIÓN 66
ILUSTRACIÓN 4-9: MÓDULO DE CONFIGURACIÓN 69
ILUSTRACIÓN 4-10: MÓDULO DE LOG 71
ILUSTRACIÓN 4-11: MÓDULO DE IA 76
ILUSTRACIÓN 8-1: PLANIFICACIÓN PFC 107
ILUSTRACIÓN II-1: MENÚ INICIAL - MAGIC QUEST 113
ILUSTRACIÓN II-2: MODO 1 JUGADOR – MAGIC QUEST 114
ILUSTRACIÓN II-3: MODO VS – MAGIC QUEST 115

Índice de tablas

Tabla 4-1: Fase 1 - Experimento 1a: logBoard - JRip 86
Tabla 4-2: Fase 1 - Experimento 1b: logBoard – J48 87
Tabla 4-3: Fase 1 - Experimento 2a: log1Exchange - JRip 87
Tabla 4-4: Fase 1 - Experimento 2b: log1Exchange – J48 87
Tabla 4-5: Fase 1 - Experimento 3a: log2Exchange - JRip 88
Tabla 4-6: Fase 1 - Experimento 3b: log2Exchange – J48 88
Tabla 4-7: Fase 2 - Experimento 1a: JRip – Inicial 89
Tabla 4-8: Fase 2 - Experimento 1b: J48 - Inicial 90
Tabla 4-9: Fase 2 - Experimento 2a: JRip – Vida menos Maná 91
Tabla 4-10: Fase 2 - Experimento 2b: J48 – Vida menos Maná 91
Tabla 4-11: Fase 2 - Experimento 3a: JRip – Eliminando las magias 93
Tabla 4-12: Fase 2 - Experimento 3b: J48 – Eliminando las magias 94
Tabla 4-13: Fase 2 - Experimento 4a: JRip – Eliminando los manas 96
Tabla 4-14: Fase 2 - Experimento 4b: J48 – Eliminando los manas 96
Tabla 4-15: Fase 2 - Experimento 5a: JRip – Eliminando magias oponente ... 98
Tabla 4-16: Fase 2 - Experimento 5b: J48 – Eliminando magias oponente 98
Tabla 5-1: Resultados IA – Jugador JRip Vs Jugador aleatorio 102
Tabla 8-1: Costes de personal.....108
Tabla 8-2: Costes totales 109

1 Introducción

La industria de los videojuegos se encuentra en un momento de gran esplendor, experimentando una gran evolución y crecimiento, especialmente en la última década. Tanto es así que a pesar de la crisis económica actual en la que nos encontramos, se trata de un sector que continúa incrementando sus ventas, superando en ingresos a la industria musical y cinematográfica juntas.

Se trata, por tanto, de una industria que puede aportar grandes beneficios económicos por lo que cualquier país desarrollado debería tenerla en cuenta actualmente. Actualmente España está comenzando a tomar medidas que apuntan a dicha dirección, pasando a considerar los videojuegos como industria cultural a fecha de 25/03/2009. Se espera que dicha medida se transforme en una serie de ventajas fiscales y ayudas en forma de subvenciones como disfrutan, por ejemplo, los productores cinematográficos. El objetivo de esta medida es el de favorecer el desarrollo y crecimiento de esta industria en nuestro país, reportando finalmente mayores beneficios económicos.

La evolución de esta industria, ligada siempre al avance de la tecnología, es más que notable. En sus poco más de 50 años de vida se ha pasado de pequeños desarrollos caseros durante algunos días o semanas con no demasiados recursos, a grandes multinacionales cuyo desarrollo de videojuegos puede alargarse durante años y sus presupuestos pueden llegar a ascender a decenas de millones de Euros.

Paralelamente, en los últimos años, ha proliferado una gran comunidad de desarrolladores no profesionales, muchas veces motivados por razones económicas, ya que pueden ver el desarrollo de videojuegos caseros como un medio para acceder al sector y llevarse un pellizco del mismo; pero otras veces simplemente por el hecho de aportar su granito de arena al sector. Las grandes compañías de videojuegos han detectado este creciente interés por el desarrollo y han decidido apoyar e impulsar el crecimiento del mismo. Gracias a ello, las compañías han distribuido *kits de desarrollo*[1] (SDK) gratuitos que permitan la creación de aplicaciones y juegos para sus respectivas plataformas. Adicionalmente las propias compañías también han establecido un medio para la publicación y distribución del producto resultante, remunerando a su desarrollador con un porcentaje del precio de venta por cada copia vendida del mismo.

Así pues, en los últimos años ha surgido ***Xbox Live Arcade*** (2004) por parte de *Microsoft*, ***WiiWare*** (2009) por parte de *Nintendo* e incluso ***App Store*** (2008) para *iPhone* por parte de *Apple* como medios de distribución de las principales compañías de videojuegos para desarrolladores no profesionales y profesionales independientes con pequeños presupuestos.

[1] Un **kit de desarrollo de software** o **SDK** (siglas en inglés de **S**oftware **D**evelopment **K**it) es generalmente un conjunto de herramientas de desarrollo que le permite a un programador crear aplicaciones para un sistema concreto.

Gracias a todo ello, surge la idea de este proyecto, el desarrollo de un videojuego con *XNA*[2]. XNA es un conjunto de herramientas con un entorno de ejecución administrado proporcionado por Microsoft que facilita el desarrollo de videojuegos para las plataformas Xbox 360, Zune y Windows. Se ha elegido esta herramienta para la realización de este proyecto imponiéndose al resto de posibilidades por diversas razones:

- Se trata de un kit de desarrollo **completamente gratuito**.
- **Facilidad de uso** en comparación con otras herramientas.
- Existe una **gran cantidad de información** y código libre de aplicaciones y otros videojuegos creados con XNA.
- **La comunidad de desarrolladores de XNA es muy grande**, por lo que será más sencillo y rápido la resolución de dudas y problemas si los hubiera.

Una vez establecida la herramienta y la plataforma de desarrollo el siguiente paso es determinar el juego que se pretende construir. Este proyecto está inspirado en el videojuego *Puzzle Quest*[3], un juego que debido a su éxito es multiplataforma, aunque inicialmente fue lanzado para las consolas *Nintendo DS* y *Sony PSP* en marzo de 2007, desarrollado por la compañía *Infinite Interactive*.

Se trata de un juego que combina acertadamente elementos tipo **puzzle** con elementos de **estrategia y rol**; basándose en el juego de tablero *Bejeweled*[4] pero destinando su uso para simular un combate 2 jugadores por turnos, e incorporando diferentes personajes y características, elementos coleccionables y magias.

Ilustración 1-1: Puzzle Quest Original

[2] XNA (**X**NA's **N**ot **A**cronymed, XNA no es un acrónimo) es una API desarrollada por Microsoft para el desarrollo de videojuegos para las plataformas Xbox 360, Zune y Windows.

[3] Véase la página http://en.wikipedia.org/wiki/Puzzle_Quest para más info. sobre Puzzle Quest.

[4] Véase la página http://en.wikipedia.org/wiki/Bejeweled para más info. sobre Bejeweled.

Del mismo modo que se establece en *Puzzle Quest*, el videojuego que se pretende desarrollar para este proyecto simula un combate 2 jugadores por turnos a partir del juego de tablero *Bejeweled*. Se ha considerado de competencia dentro de este proyecto el desarrollo del comportamiento del jugador automático, llevándolo a cabo mediante el uso de técnicas de *Inteligencia Artificial*[5] (IA).

La mayoría de los videojuegos actuales requieren, en mayor o menor medida, el uso de la Inteligencia Artificial, pudiendo utilizarse gran variedad de técnicas para el desarrollo de la misma. La Inteligencia Artificial se trata de un **factor clave a la hora de desarrollar un videojuego** ya que influye directamente en la jugabilidad del mismo, pudiendo convertir un título brillante en el resto de apartados, en un título molesto y tedioso para jugar en el caso de que la IA no esté a la altura.

Existen varias técnicas que permitirían desarrollar el comportamiento del jugador automático para el proyecto, por lo que será necesario llevar a cabo un estudio para valorar cada una de las técnicas a la hora de determinar cuál será la elegida para el desarrollo de la misma dentro de este proyecto.

Con todo ello, el objetivo principal del proyecto es la **creación de un videojuego para PC mediante la tecnología *XNA* de *Microsoft*, inspirado en el actual videojuego *Puzzle Quest***. De esta forma se pretende culminar la carrera de Ingeniería en Informática con especialización en Inteligencia Artificial con un PFC que supone una motivación personal, permitiendo de este modo adquirir los conocimientos necesarios para introducirse en el mundo del desarrollo de videojuegos, y realizando un aporte a dicha comunidad de forma que se pueda continuar el desarrollo del mismo posteriormente.

[5] La **Inteligencia Artificial (IA)** aplicada a las ciencias de la computación se define como la creación de sistemas que tienen como fin lograr la resolución de problemas para los cuales sería preciso el razonamiento humano.

2 Estado de la cuestión

Antes de introducirnos en el desarrollo del proyecto es necesario comprender y reflexionar sobre la historia y el estado del ámbito en el cual se va a trabajar en este proyecto. Para ello, en primer lugar se realizará un breve repaso de la historia y evolución de los videojuegos y su industria; llegado al presente se expondrán las distintas posibilidades actuales para el desarrollo no profesional de videojuegos, valorando dichas alternativas y argumentando la decisión sobre la plataforma final en la que se llevará a cabo el desarrollo de este proyecto. Seguidamente se presentarán y describirán algunas de las características del videojuego *Puzzle Quest*, el cual ha servido de inspiración para este proyecto. Por último, dado que en este proyecto también se pretende abarcar el desarrollo del comportamiento del jugador automático mediante técnicas de inteligencia artificial, se realizará un breve estudio de diferentes técnicas posibles, determinando finalmente la técnica utilizada y los motivos de dicha decisión.

2.1 Historia y evolución de los videojuegos

La historia de los videojuegos comenzó en 1947 con el sistema electrónico de juego denominado ***Lanzamiento de misiles***[6]. Sus creadores fueron Thomas T. Goldsmith y Estle Ray Mann, los cuales crearon y **patentaron la idea de videojuego**. Este sistema se basaba en las pantallas de radar que usaba el ejército en la entonces reciente segunda guerra mundial. Permitía a los jugadores ajustar la velocidad y la curva de disparo, pero los objetivos estaban sobreimpresionados, no había movimiento de video en la pantalla. Es por estas razones por lo que, a pesar de todo, actualmente no se le considera un videojuego al no cumplir las características básicas de los mismos, pero **es considerado como un precedente**.

Cinco años después, en 1952, Alexander Sandy Douglas presenta su tesis de doctorado en matemáticas en la Universidad de Cambridge (Inglaterra) sobre la interactividad entre seres humanos y computadoras. La tesis incluye el código del primer juego gráfico con constancia segura. Se trata de una versión del "***Tres en Raya***" (Tic Tac Toe) para una computadora EDSAC[7], diseñada y construida en esa misma universidad. El programa tomaba las decisiones correctas en cada momento del juego según el movimiento realizado por el jugador, que lo hacía mediante un dial telefónico de rueda que incorporaba la computadora EDSAC. **Este juego suele ser tratado como otro precedente**, ya que no se le considera realmente un videojuego, sino un juego gráfico por ordenador, al no existir video en movimiento.

[6] Véase la página http://indicelatino.com/juegos/historia/origenes/ para más información sobre el 'videojuego' Lanzamiento de misiles y otros considerados como los orígenes del videojuego.

[7] La **EDSAC** *(acrónimo proveniente de la frase **E**lectronic **D**elay **S**torage **A**utomatic **C**alculator)*, fue una antigua computadora británica (una de las primeras computadoras creadas).

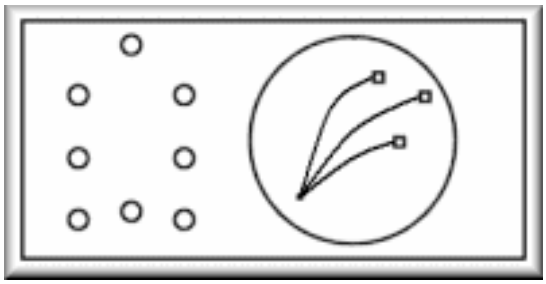

Ilustración 2-1: Lanzamiento de misiles

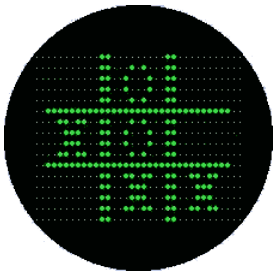

Ilustración 2-2: Tres en Raya

En 1957 William Higginbotham creó, sirviéndose de un programa para el cálculo de trayectorias y un osciloscopio, ***Tennis for Two***: un simulador de tenis para entretenimiento de los visitantes del Brookhaven National Laboratory. El juego constaba de una línea horizontal que era el campo de juego y otra pequeña vertical en el centro del campo representando la red. Los jugadores debían elegir el ángulo en el que salía la bola y golpearla. Aunque existe cierta controversia, se puede considerar como **el primer videojuego de la historia**, permitiendo además el juego entre dos oponentes humanos.

Cuatro años más tarde, en 1961, Steve Russell, un estudiante del Instituto de Tecnología de Massachussets, dedicó seis meses a crear un juego para una computadora PDP-1[8]usando gráficos vectoriales: ***Spacewar!***. El juego era para dos jugadores, cada uno manejaba una nave espacial e intentaba disparar a la otra, además había en la pantalla una estrella cuya gravedad atraía a las naves hasta destruirlas si las alcanzaba. **Se trata del primer videojuego para ordenador de la historia.**

Atari*, fundada en 1972** en los Estados Unidos por Nolan Bushnell y Ted Dabney puede ser considerada la fundadora de la industria del videojuego. Se encargó de desarrollar y comercializar una versión de *Tennis for Two* bajo el nombre de ***Pong, el cual apareció en las primeras máquinas recreativas.

Ilustración 2-3: Spacewar

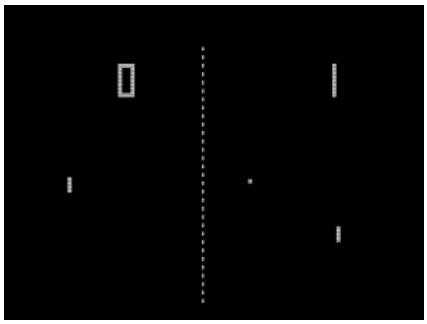

Ilustración 2-4: Pong

[8] **PDP-1** (***P****rogrammed* ***D****ata* ***P****rocessor-****1***) fue el primer computador en serie PDP de la Digital Equipment, producida por primera vez en 1960

En 1972 también salió al mercado **la primera videoconsola de la historia**, la ***Magnavox Odyssey***. Se trataba de una consola tan primitiva que los jugadores tenían que anotar sus puntos en un papel ya que el aparato carecía de memoria alguna. Tres años después **contraatacó *Atari* creando su primera consola doméstica basándose en su popular videojuego *Pong***, para crear un chip y una consola que permitía jugar en casa, en cualquier aparato de televisión. Este sistema, que sólo permitía jugar a dicho juego, supuso una verdadera revolución del mercado en las navidades de 1975 aunque a medio/largo plazo no tuvo una gran acogida.

Dos años más tarde, en 1977, Atari consiguió el éxito que perseguía con su nueva consola, la ***Atari 2600***. Con tan solo 8 bits de potencia se considera una revolución en el mundo de los videojuegos gracias a innovaciones como poder cambiar de juego mediante el nuevo **sistema de cartuchos**. El éxito de dicha consola la mantuvo liderando el mercado como la consola más vendida durante casi una década, ensombreciendo a la cada vez mayor competencia que comenzaba a surgir, como fueron la *Magnavox Odyssey 2* y la *Mattle Intellivision*.

Tuvo que ser ***Nintendo* en 1985, con su *NES*** (Nintendo Entertainment System) la que destronara a *Atari 2600* de su hegemonía en un periodo de crisis para el sector en el que parecía que se había tocado techo sin nada realmente nuevo ni sorprendente. *NES* tiene, al igual que su rival, 8 bits de potencia pero incorpora varias innovaciones como el mando de control, denominado pad. Pero sin duda el punto fuerte de *Nintendo* radica en la aparición del videojuego más famoso de la historia: ***Super Mario Bross***.

Nintendo, respaldada por la genialidad de la creación de sus juegos dio un giro de tuerca, consiguiendo pronto ser la primera videoconsola exitosa para su fabricante. Además jugó a su favor la exclusividad y el monopolio de sus juegos que obligaba a las empresas desarrolladores a mantenerse fieles a ella y no ceder sus creaciones a otras plataformas competentes en el mercado.

Un año más tarde, **en 1986, *Sega*** con su ***Master System*** de 8 bits de potencia da comienzo una batalla por el dominio del mercado que poseía Nintendo. No tuvo gran impacto en el mercado nipón, pero sí en el mercado americano y el europeo, donde llegó a conseguir un notable éxito llegando a igualar en ventas a la *NES*.

No obstante, tan solo dos años más tarde, **en 1988, *Sega*** decidió apostar por una **nueva máquina de 16 bits, la Mega Drive**, intentando obtener mayores ventas y popularidad ante una supuesta mejora en la calidad técnica del aparato. Tras un inicio complicado finalmente consigue un buen éxito de ventas durante bastante tiempo. Con esta videoconsola *Sega* se labró un buen prestigio y estableció, de la misma forma que sucedía con *Mario* en *Nintendo*, un **estandarte y mascota: *Sonic*** el puercoespín ultrasónico.

En 1992 Nintendo decide competir en igualdad de condiciones con Sega y **comercializa su propia consola de 16 bits, la Super Nintendo**, que consigue también un gran éxito de ventas. A partir de ese momento la estrategia de ambas compañías se bifurca de modo que se enfocó en sacar más y mejores juegos mientras Sega se centró en sacar periféricos para su consola en un intento de potenciarla. Durante su vida útil, fueron vendidas **65 millones de *Super Nintendo* y 21 millones de *Mega Drive***.

Ilustración 2-5: Mega Drive

Ilustración 2-6: Super Nintendo

En 1994 Sega vuelve a sacar una nueva consola, la ***Sega Saturn*** de 32 bits, para tratar de adelantarse a la cercana llegada de la *PayStation* de *Sony*. Esta nueva consola de *Sega* no tuvo buena acogida, ya que los usuarios estaban desencantados con la compañía por el gasto que suponía la compra de periféricos y nuevas consolas cada poco tiempo.

Semanas más tarde del lanzamiento de *Sega Saturn*, pero **aún en 1994**, fue puesta a la venta en Japón la ***PlayStation* de *Sony*** (1995 en EEUU y Europa), también conocida como *PSX*, con 32 bits de potencia. El éxito de la consola de Sony fue rotundo, con unas ventas estimadas a día de hoy de **102 millones de unidades**. La clave de su éxito, ayudado en gran parte por la masiva piratería de los videojuegos para dicha plataforma, residía en que, al contrario que sus competidoras, la *PlayStation* ofrecía un **tipo de juego mucho más adulto y serio**, por lo que introdujo en el mundo de las consolas a millones de personas de todas las edades.

En 1996, Nintendo comercializó la ***Nintendo 64*** en Japón y en EEUU (1997 en Europa), una consola con 64 bits de potencia y juegos en formato cartucho, siguiendo la estela de sus predecesoras. Se trataba de una consola ideada para juegos en 3 dimensiones, que poseía bastantes aspectos innovadores. El principal de ellos era el mando, que incorporaba un mayor número de botones, botones superiores, un gatillo y un **stick analógico**. Precisamente este último permitía un control mucho más preciso, ideal para los juegos en 3 dimensiones para los que estaba pensada la consola. El videojuego ***Super Mario 64*** mostró el potencial de dicha consola, causando gran expectación inicial, pero ensombrecida finalmente por la poderosa *PlayStation*. Aun así se vendieron alrededor de **32 millones de unidades** y ambas consolas tuvieron una **coexistencia pacífica** debido a que estaban dirigidas a un público muy distinto.

Ilustración 2-7: PlayStation

Ilustración 2-8: Nintendo 64

Durante este tiempo de reinado indiscutible de *Sony*, ***Sega*** preparaba en silencio minuciosamente la que en 1998 sería **su última videoconsola: *Dreamcast*, un potente hardware de 128 bits** con unos gráficos muy superiores a lo visto hasta ese momento, un precio asequible, un buen catálogo inicial de juegos, posibilidad por primera vez de jugar online y con 4 puertos para conectar 4 mandos. Sin embargo, debido a la ya comentada falta de confianza en la compañía, y a la inminente llegada de la esperada *PlayStation 2*, poca gente se decidió por ella, vendiendo **tan solo 11 millones de unidades** y llevando a Sega poco después a tomar la decisión de **abandonar la industria de hardware para consolas**, dedicándose únicamente al desarrollo de software para otras compañías.

PlayStation 2* de *Sony*, en el año 2000, era todo lo que se podía esperar**, tenía los mismos 128 bits de potencia, almacenamiento en DVD y una gran cantidad de compañías apoyándola. Estos factores, y la enorme cantidad de usuarios fieles a la compañía, consiguieron que fuera **la consola de sobremesa más vendida de la historia** (aún en producción), con unas ventas de **más de 125 millones de unidades**. **Sus competidoras** en esta generación de consolas fueron la ***Nintendo GameCube* en 2001** en Japón y EEUU (2002 en Europa) y la ***Xbox* en 2001 en EEUU** (2002 en Japón y Europa), ésta última la primera incursión de *Microsoft* en el mundo de las consolas de sobremesa. Se estima que se vendieron **unos 22 millones de *GameCube* y unos 24 millones de *Xbox.

La siguiente generación de consolas es la actual y está formada por ***Xbox 360* de *Microsoft*, lanzada en 2005** y con unas ventas de 40 millones de unidades; ***PlayStation 3* de *Sony*, lanzada en 2006** y con unas ventas de 35 millones de unidades; y ***Wii* de *Nintendo*, lanzada también en 2006**, a la que ya a día de hoy se le puede considerar como **la gran triunfadora** de esta generación, con unas ventas hasta la fecha de 67 millones de unidades.

La *Wii* de *Nintendo* ha marcado una nueva evolución en el mundo de los videojuegos gracias a una transformación de la experiencia de juego debido a su **revolucionario mando de control, el *Wiimote***. Su simplicidad de control, basada en la **captación del movimiento** gracias a la incorporación de acelerómetros, ha permitido eliminar una gran cantidad de barreras en el mundo de los videojuegos, **eliminando las complicaciones** de tener que apretar muchos

botones. De esta forma *Nintendo* ha conseguido **expandir el mercado** gracias al interés de muchas personas que nunca antes habían tenido contacto con los videojuegos, como puede ser el caso de las **personas de media y avanzada edad**, así como las **mujeres** que antes eran una minoría.

Tal es el éxito de *Nintendo Wii*, que tanto *Sony* como *Microsoft* han decidido desarrollar mecanismos que imiten el estilo de juego "sin botones". Este es el caso de ***PlayStation Move* por parte de *Sony***, y de ***Kinect* por parte de *Microsoft***. Ambas tecnologías están comenzando a ser lanzadas al mercado en la actualidad, junto con su respectivo catálogo de videojuegos.

Ilustración 2-9: Wii, PlayStation 3, Xbox 360

2.2 Herramienta de desarrollo: XNA

2.2.1 Introducción

***XNA* es un conjunto de herramientas**, desarrolladas por Microsoft, que proporcionan una API[9] **para el desarrollo de videojuegos** para las plataformas *Xbox 360, Zune y Windows*. Técnicamente es un Marco de Trabajo (Framework[10]), basado en *DirectX*[11] y *.NET Framework 2.0* y al igual que él, *XNA* corre sobre *CLR*[12], aunque en una implementación que provee un manejo optimizado para la ejecución de videojuegos.

[9] Una **interfaz de programación de aplicaciones** o **API** (del inglés ***A**pplication **P**rogramming **I**nterface*) es el conjunto de funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

[10] Un **framework** es una estructura conceptual y tecnológica de soporte definida, normalmente con módulos de software concretos, con base en la cual otro proyecto de software puede ser organizado y desarrollado. Típicamente, puede incluir soporte de programas, bibliotecas y un lenguaje interpretado entre otros programas para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

[11] **DirectX** es una colección de API creadas y recreadas para facilitar las complejas tareas relacionadas con multimedia, especialmente juegos y vídeo en la plataforma Microsoft Windows.

[12] El **Common Language Runtime** o CLR (Lenguaje común en tiempo de ejecución) es el componente de máquina virtual de la plataforma .Net de Microsoft.

El *XNA Framework*, por lo tanto, permite simplificar y hacer más intuitivo el uso de las librerías nativas *DirectX* y, en consecuencia, **simplifica de manera notable la programación de videojuegos** permitiendo a los desarrolladores de juegos concentrase más en el contenido y la experiencia de juego.

Para trabajar con *XNA Framework* es necesario un entorno de desarrollo que, en el caso de este proyecto concreto, ha sido el ***XNA Game Studio 3.0***. Se trata de un entorno de desarrollo integrado (IDE) que **reúne las bibliotecas de *XNA*** para *Xbox 360, Zune y Windows*, **herramientas de gestión de contenido** (el *XNA Content Pipeline*), ***starter kits***[13] para el rápido desarrollo de géneros de juego específicos como son plataforma, estrategia en tiempo real y FPS[14], e **integración con una versión gratuita del IDE *Visual Studio 2008*** recortado para su uso sólo con el lenguaje de programación C#.

Los desarrolladores que quieran trabajar con *XNA* deben **escribir el código en un lenguaje como C# o VB.Net** en un IDE como Visual Studio 2008. En tiempo de compilación un compilador .NET convierte el código desarrollado a un **código intermedio** (bytecode) llamado ***Common Intermediate Languaje***[15]. En tiempo de ejecución, el compilador en tiempo de ejecución (Just-in-time compiler) del *CLR* convierte el código *CIL* en código nativo para el sistema operativo. Alternativamente, el código *CIL* es compilado a código nativo en un proceso separado anterior a la ejecución. Esto acelera las posteriores ejecuciones del software debido a que la compilación de *CIL* a nativo ya no es necesaria.

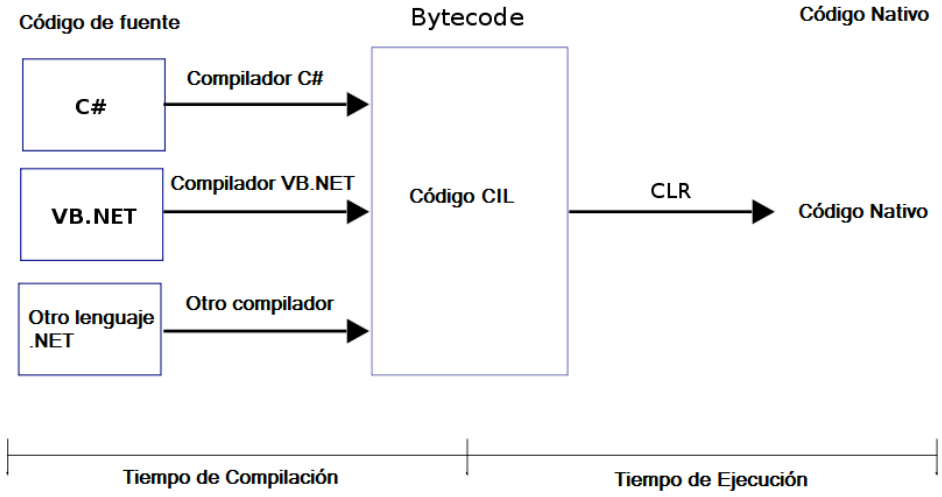

Ilustración 2-10: Compilación de .NET

[13] Los **starter kits** son juegos desarrollados por la propia Microsoft para XNA con su código y un documento de ayuda donde describen el comportamiento y funcionamiento del juego y su código y una serie de ideas para que un desarrollador que quiera iniciarse en XNA intente desarrollarlas.

[14] Los **videojuegos de disparos en primera persona**, también conocidos como **FPS** (siglas en inglés de *first-person shooter*), es un género de videojuegos y subgénero de los videojuegos de disparos que se desarrolla desde la perspectiva del personaje protagonista.

[15] **Common Intermediate Language** (CIL, anteriormente llamado **Microsoft Intermediate Language** o MSIL) es el lenguaje de programación legible por humanos de más bajo nivel en el Common Language Infrastructure y en el .NET Framework.

Para ilustrar todo lo explicado en los párrafos anteriores, en el siguiente esquema se muestra una estructura de las diferentes capas de XNA. En él se puede observar, visto de arriba abajo, como el *XNA Game Studio* utiliza la funcionalidad del *XNA Framework*, y éste a su vez se basa en el *.NET Framework*. Por último están las plataformas a las que van destinadas nuestras aplicaciones, *Windows*, *Xbox 360* o *Zune*.

Ilustración 2-11: Capas de XNA

2.2.2 Breve historia de XNA

XNA está enmarcado dentro de la **tecnología *.NET*** de *Microsoft*. El nacimiento de dicha tecnología se produce **en 2001**, estableciendo una batalla en el mundo de software con la plataforma *Java* de *Sun Microsystems*, nacida en 1991.

.NET es un proyecto de Microsoft para crear una **nueva plataforma de desarrollo de software** con énfasis en la transparencia de redes, con **independencia de plataforma** y que permite un rápido desarrollo de aplicaciones. Basado en esta plataforma, Microsoft intenta desarrollar una estrategia horizontal que integre todos sus productos, desde el sistema operativo hasta las herramientas de mercado.

Java y .NET comparten una concepción similar y un mismo modelo de desarrollo, basándose en la independencia de la plataforma mediante la abstracción del hardware subyacente por medio de una ***máquina virtual***[16] a pila, orientada a objetos y con distintos servicios comunes a toda la aplicación, como la gestión automática de memoria, que permite al desarrollador olvidarse de los aspectos más rutinarios y centrarse en la materialización de sus ideas.

[16] Una **máquina virtual** es un software que emula a una computadora y puede ejecutar programas como si fuese una computadora real.

La plataforma ***.NET***, sin embargo, ha sido **ideada para ser empleada por diferentes compiladores de diferentes lenguajes**, creados o no por Microsoft. Para ello, se han establecido una serie de **normas**, a las que se denomina ***CLS***[17], que se deben usar para asegurar la interoperatividad entre lenguajes. De esta forma, por ejemplo, se especifica como se debe hacer uso de mayúsculas y minúsculas en el nombrado de atributos y métodos ya que lenguajes como VB.NET no los distinguen.

Un año más tarde, **en 2002, Microsoft creó *Managed DirectX* (*MDX*)**, una serie de bibliotecas para la plataforma *.NET* que permiten a través de la misma el **acceso a la API de *DirectX***. De esta forma los desarrolladores pueden acceder a través de .NET a las numerosas clases de representación de gráficos en 3D (*Direct3D*) y otras APIs de *DirectX* de una manera mucho más fácil, de manera orientada a objetos.

XNA surge en **diciembre de 2006**, considerándose como una **evolución de *MDX*** hasta el punto de anunciarse el cese del soporte de esta última. *XNA* va más allá de *MDX* y, lejos de ser un simple conjunto de bibliotecas, **se establece como un Marco de Trabajo** y, como tal, ofrece una serie de tipos destinados a ser ejecutados dentro de un entorno definido. Por ejemplo, existe un método que permite al usuario introducir lo que se quiere dibujar por pantalla; el usuario únicamente debe centrase en qué introducir, desligándose por completo de las tareas como llamar a ese método cada cierto intervalo de tiempo o inicializar el dispositivo gráfico. Gracias a todo ello, *XNA* consigue un mayor nivel de abstracción, simplificando en mayor medida la labor del desarrollador.

El entorno de desarrollo para trabajar con *XNA* recibió el nombre de ***XNA Game Studio Express***, que se trataba de un entorno de desarrollo integrado (IDE) que reúne las bibliotecas de *XNA* para *Xbox 360 y Windows*, herramientas de gestión de contenido (el *XNA Content Pipeline*), *starter kits* e integración con una versión gratuita del IDE *Visual Studio 2005* recortado para su uso sólo con el lenguaje de programación C#.

Tan sólo 4 meses más tarde, en **marzo de 2007**, *Microsoft* lanza la primera **actualización** de *XNA*, denominada ***XNA Game Studio Express 1.0 Refresh***, que incluía algunas mejoras como la posibilidad de hacer paquetes de instalación que facilitaban el despliegue de videojuegos desde *PC* a *Xbox 360*.

En **diciembre de 2007**, apenas un año después de su primera versión, se lanza ***XNA Game Studio 2.0***, con mejoras como una API de red usando *Xbox Live*[18] en *Windows* y *Xbox 360*, y mejoras en el dispositivo de manipulación.

[17] **CLS** *(**C**ommon **L**anguage **S**pecification)* es un conjunto de especificaciones que determina las reglas necesarias para crear el código CIL compatible con el CLR.

[18] **Xbox live** es el nombre del servicio online de Microsft que da soporte a los juegos multijugador de la Xbox 360 y la antigua Xbox además de las plataformas con Windows.

XNA Game Studio 3.0 aparece un año después, en **octubre de 2008**, que pasa a integrarse con IDE *Visual Studio 2008*, e incluye mejoras como la posibilidad de producción de juegos para la **plataforma *Zune*** y agrega soporte de la comunidad de *Xbox Live*. Otra característica nueva que se incluye en esta versión es la posibilidad de agregar por parte de los creadores una **función de prueba** para sus juegos.

En **noviembre de 2009** aparece una nueva versión de esta última, ***XNA Game Studio 3.1***. Entre las mejoras introducidas está la inclusión de una **API para el soporte de la reproducción de video y mejoras en la API de audio**.

Por último, en **marzo de 2010** se ha lanzado la última versión disponible hasta la fecha, ***XNA Game Studio 4.0***, que pasa a integrarse con IDE *Visual Studio 2010*, cuya principal novedad es el soporte para la plataforma ***Windows phone***.

2.2.3 Elección de XNA para el desarrollo del proyecto

XNA proporciona un conjunto de herramientas para el desarrollo de videojuegos para las plataformas *Xbox 360, Zune y Windows.* No obstante, **no se trata de la única alternativa**, ya que en los últimos años han surgido una gran cantidad de herramientas de desarrollo para distintas plataformas.

Microsoft no ha sido la única compañía que ha detectado el creciente interés por el desarrollo de no profesionales y así, otras grandes compañías del mundo de los videojuegos han adoptado posturas similares. Este es el caso, por ejemplo, de **Nintendo con WiiWare**, que permite el desarrollo de videojuegos para Wii por parte de desarrolladores no profesionales y profesionales independientes con pequeños presupuestos.

También hay que destacar el caso de ***Apple*** con los juegos y aplicaciones **para sus plataformas *iPhone* e *iPod Touch***. La gran aceptación que ha tenido el *iPhone* desde su lanzamiento, unido a la **distribución gratuita de su *SDK*** por parte de la compañía, así como las facilidades para distribuir y vender las aplicaciones y juegos desarrollados ha favorecido la proliferación del número de desarrolladores para dicha plataforma. Esta **opción fue una de las principales a la hora de elegir plataforma** de desarrollo para el proyecto, pero finalmente se descartó porque en la fecha en la que se comenzó la realización del proyecto para poder desarrollar **era necesario poseer un ordenador *Mac***, y también es **necesario poseer un *iPhone* o *iPod Touch*** si se quiere probar el producto resultante en la plataforma final.

Por último en lo que respecta al propio **desarrollo de juegos para PC**, existen otras herramientas como ***DarkBasic*, *Panda3D*, *Torque3D* o *Biltz3D***. Todas ellas tienen una serie de pros y contras pero, finalmente, por las razones que se expondrán a continuación, se decidió utilizar la herramienta *XNA* de *Microsoft*.

Una vez vistas las distintas posibilidades existentes para el desarrollo del videojuego, se ha elegido *XNA* para la realización de este proyecto imponiéndose al resto de posibilidades por diversas razones:

- Se trata de un kit de desarrollo **completamente gratuito**: al contrario que otras de las plataformas anteriormente mencionadas, como *Torque3D* o *Biltz3D*; *XNA* es completamente gratuita. Únicamente sería necesario una subscripción a *XNA Creators Club* a razón de 50€ por 4 meses o 100€ por año en caso de estar interesado en distribuir el juego para la plataforma Xbox 360.
- **Se disponía de todas las herramientas hardware necesarias** para llevar a cabo el desarrollo y las pruebas: Este punto **descartó** una de las principales opciones, la de ***iPhone* e *iPod Touch*** ya que en la fecha en la que se comenzó la realización del proyecto para poder desarrollar era necesario poseer un ordenador *Mac*, y también es necesario poseer un *iPhone* o *iPod Touch* si se quiere probar el producto resultante en la plataforma final.
- **Permite el desarrollo de juegos para** *Xbox 360, Zune y* ***Windows***: Una de las plataformas preferidas para las que se pretende desarrollar el proyecto era el PC, y XNA está completamente orientada a dicha plataforma. Además, gracias a las características de XNA es bastante sencilla, si se considerara oportuna, su conversión para otras plataformas como son Xbox 360 o Zune.
- **Facilidad de uso** en comparación con otras herramientas: *XNA* fue desarrollado por *Microsoft* expresamente para el desarrollo de juegos por parte de desarrolladores no profesionales. Por ello resulta **sencilla** de usar en comparación con otras de las opciones, además de estar ampliamente **testada contra fallos**, ser **robusta** y estar **en continúa revisión y actualización**.
- Existe una **gran cantidad de información** y código libre de aplicaciones y otros videojuegos creados con XNA: Este hecho permite que la curva de aprendizaje para esta plataforma sea más rápida y, además, los desarrollos realizados gozarán de una mayor calidad.
- **La comunidad de desarrolladores de XNA es muy grande**: Actualmente es una de las plataformas más usadas entre todas las comentadas anteriormente, con numerosas comunidades y foros de ayuda; por lo que será más sencillo y rápido la resolución de dudas y problemas si los hubiera.
- **Propiedad de Microsoft**: Se trata de un gran punto a favor tener como respaldo a una gran empresa como Microsoft, la cual además, se está volcando en el uso y extensión de dicha plataforma, dando soporte, revisándola y mejorándola constantemente.

2.2.4 Componentes a instalar para usar XNA y requisitos mínimos

En este apartado únicamente se enumeran los distintos componentes a instalar para poder desarrollar con *XNA* o bien para poder ejecutar las aplicaciones desarrolladas. Adicionalmente también se especifican los requisitos mínimos para que dicha herramienta funcione de manera óptima. Se recomienda consultar el Anexo I: Componentes a instalar para usar XNA y requisitos mínimos para obtener una descripción más detallada.

Los componentes necesarios para **desarrollar** con *XNA* son los siguientes:

- *XNA Game Studio 3.0*.
- *Visual C# 2008 Express Edition*.
- *.NET Framework 3.5*.

Los componentes necesarios para poder **ejecutar** aplicaciones desarrolladas con *XNA* son los siguientes:

- *XNA Framework Redistributable 3.0*.
- *DirectX 9.0 Runtime*.
- *.NET Framework 3.5*.

Para un funcionamiento óptimo en el desarrollo y ejecución son necesarios los siguientes **requisitos mínimos**:

- 512 MB de RAM.
- Tarjeta gráfica que soporte *DirectX 9.0* o superior, compatible con shaders 1.1 como mínimo. Son compatibles las tarjetas de la serie FX de *Nvidia GeForce* y las *ATI Radeon 9500* o superiores.
- Espacio libre en disco de al menos 2 GB.
- Sistema Operativo *Windows XP SP3*, *Windows Vista* o *Windows 7*.
- Otros requisitos hardware dependen totalmente del juego desarrollado.

2.2.5 Descripción de la plantilla inicial

El desarrollo de un videojuego con *XNA* se realiza con la herramienta ***Microsoft Visual C#***, utilizando la versión 2008 Express Edition en este proyecto. Es necesario crear un nuevo proyecto e indicar el producto que se pretende desarrollar, en nuestro caso un **Windows Game 3.0** (o bien un Xbox 360 3.0), indicando el nombre del mismo y la ruta en la que se desea trabajar. Tras ello, se puede observar en el explorador de soluciones de la derecha que han aparecido dos archivos de código: *Game1.cs* y *Program.cs*. Se trata de una **plantilla inicial** de la cual que se debe partir para el desarrollo del videojuego.

Esta plantilla **se puede considerar como un juego vacío**, ya que el resultado de su ejecución sería el de una pantalla de juego, con unas determinadas dimensiones, totalmente vacía. No obstante, su ejecución ya se comporta como el de un juego desarrollado, actualizando cada cierto intervalo de tiempo la lógica del juego y refrescando la imagen que muestra por pantalla, aunque en este caso sea una pantalla vacía.

Respecto a los archivos generados, ***Program.cs*** apenas tiene unas pocas de líneas de código ya que únicamente contiene el **método main**, que es el punto de inicio donde comienza la ejecución del juego. La funcionalidad de éste es llamar al método *run* del archivo *Game1.cs*, que se encargará de crear un **continuo bucle** que permita el funcionamiento del juego, llamando a los métodos necesarios cada cierto intervalo de tiempo. No hay nada que se necesite cambiar de este archivo.

El archivo ***Game1.cs*** es la base principal del juego, que **sirve como base para añadir nuevo código y desarrollar el juego**. Esta clase hereda de `Microsoft.Xna.Framework.Game`, que se trata de una clase del propio *XNA Framework* orientada a un desarrollo sencillo y eficiente del videojuego. Será necesario sobrescribir los principales métodos de esta clase aportando las funcionalidades deseadas para el videojuego concreto a desarrollar.

La **estructura inicial de *Game1.cs*** está formada por un par de variables/objetos, el constructor y cinco métodos. Esta estructura tiene las siguientes características:

- Las variables/objetos ***`graphics`*** **y** ***`spriteBatch`*** **tienen como función el tratamiento gráfico del juego**. `Graphics` actúa como manejador de la tarjeta gráfica. `SpriteBatch` es usado para los juegos 2D, como es el caso de este proyecto, con la funcionalidad de permitir dibujar las texturas 2D, también llamadas *sprites*.
- El ***`constructor`*** es llamado una vez al inicio de la ejecución y se usa para crear la clase y cargar algunas variables necesarias para el *XNA Framework*.

- El método ***`Initialize`*** se llama una única vez al inicio del juego y permite **inicializar los datos y recursos necesarios para ser ejecutado**. Se trata de la región donde se puede hacer la petición de servicios y la carga de contenido NO gráfico.
- El método ***`LoadContent`*** se llama una única vez al inicio del juego y permite **cargar el contenido gráfico del juego**.
- El método ***`UnloadContent`*** permite **eliminar el contenido gráfico cargado** con la finalidad de liberar recursos. Es llamado automáticamente.
- El método ***`Update`*** es el encargado de **actualizar constantemente la lógica del juego** como por ejemplo la actualización del mundo, el chequeo de colisiones, la interpretación de la entrada por parte del usuario, y llevar a cabo la reproducción del audio. Este método, dentro del bucle de ejecución, será llamado aproximadamente unas 60 veces por segundo.
- El método ***`Draw`*** es el encargado de llevar a cabo la **representación gráfica del juego**, mostrando los componentes oportunos por pantalla. Este método permite encapsular los componentes del juego que se desean dibujar y mandar la información a la tarjeta gráfica del PC de modo que sea capaz de interpretarlo y mostrarlo por pantalla. Para este proyecto en concreto al tratarse de un juego 2D, será necesario utilizar el objeto *spriteBatch* para encapsular las texturas 2D que se deseen dibujar. Este método, dentro del bucle de ejecución, será llamado con la máxima frecuencia que sea posible teniendo en cuenta la capacidad de la tarjeta gráfica.

Es importante comprender la estructura y el funcionamiento de la plantilla inicial ya que servirá como base para el desarrollo del videojuego. Es en este punto donde cada desarrollador introduce los componentes característicos del juego que pretende desarrollar para obtener su producto final. Existen múltiples tutoriales donde se explican los primeros pasos para el desarrollo de un videojuego, habiendo sido de gran utilidad en el desarrollo del proyecto el tutorial *2D Series: Shooters!* de www.riemers.net para la comprensión de las funcionalidades básicas para los juegos 2D en XNA.

2.3 Videojuego Puzzle Quest

El proyecto que se pretende desarrollar está inspirado en el videojuego *Puzzle Quest*, un juego desarrollado por la compañía *Infinite Interactive* y publicado por *D3 Publisher*. Inicialmente fue lanzado para las consolas *Nintendo DS* y *Sony PSP* en marzo de 2007 y, gracias a su éxito, posteriormente fue portado a otras plataformas como son *Xbox Live Arcade*, *Wii*, *Windows*, *PlayStation 2*, *PlayStation 3* e incluso *iPhone* y versiones para móviles. De hecho gracias a la buena acogida del producto, sus creadores han desarrollado una serie de juegos basados en una fórmula similar, como puede ser el caso del juego Puzzle *Chronicles* o *Puzzle Kingdoms*. Además, recientemente ha sido lanzado su secuela, *Puzzle Quest 2*, continuando la fórmula de su predecesor.

La clave de su éxito radica en que combina acertadamente elementos tipo **puzzle** con elementos pertenecientes a los géneros de **estrategia y rol**; basándose en el juego de tablero *Bejeweled* pero destinando su uso para simular un combate 2 jugadores por turnos, e incorporando diferentes personajes y características, elementos coleccionables y magias.

El juego posee muchos elementos típicos del género de rol, ya que al comienzo de la partida **el jugador asume el papel de uno de los varios personajes**, que posee ciertas estadísticas y magias. El juego consta de un sencillo **mapa** en el cual los jugadores aceptan **misiones**, bien que forman parte de la historia principal del juego, o bien se tratan de misiones secundarias útiles para conseguir **experiencia** adicional **y oro**. También se puede conseguir experiencia y oro derrotando a los enemigos que aparecen periódicamente bloqueando el paso en los caminos que conectan los distintos lugares del mapa.

Ilustración 2-12: Puzzle Quest - Mapa

La **experiencia** obtenida, como en la mayoría de los juegos de rol, permite al personaje **subir de nivel**, potenciando sus estadísticas, permitiendo aprender nuevas magias más potentes y permitiendo comprar y usar mejores equipos. Por otra parte el **oro** obtenido puede ser utilizado para **comprar equipos** que permiten potenciar las características del personaje, ofreciendo diversos bonus y mejoras para el personaje en el combate.

Es en el **sistema de combate** en el que hace su aparición el **género puzzle**, ya que este se lleva a cabo a través de un juego de **tablero similar a *Bejeweled***. En dicho tablero el jugador y el oponente, controlado por la IA, se turnan para intercambiar horizontal o verticalmente la posición de dos casillas adyacentes de modo que se genere una línea de al menos 3 casillas iguales. De este modo las casillas se eliminan, produciendo diferentes efectos, y se desplazan hacia abajo todas las casillas que se encuentran por encima de ellas para llenar los espacios, creando nuevas en la parte superior del tablero. Si esta acción produce una nueva línea, también será eliminada produciendo lo que se denomina una cadena.

Existen **cinco tipos de casillas** en los tableros:

- **Maná** (rojo, amarillo, verde, azul): Permiten acumular maná del correspondiente color de la casilla, que será necesario para poder utilizar las magias del personaje.
- **Experiencia** (estrellas púrpuras): Añaden al personaje experiencia adicional al final de la batalla.
- **Oro**: Incrementan el dinero del personaje al final de la batalla.
- **Calaveras**: Producen daño directo al otro jugador. Algunas calaveras tienen un aura roja, que indica que producen 5 veces más daño y destruyen todas las piezas que hay alrededor cuando forman una línea.
- **Comodines**: Permiten combinar la línea con maná de cualquier otro color. El comodín además contiene un multiplicador que permite que el maná obtenido en la línea generada se multiplique por el número que contiene el comodín. Los comodines aparecen cuando un jugador genera una fila de 5 casillas.

Ilustración 2-13: Tipos de casillas en Puzzle Quest

El objetivo del combate es **derrotar al enemigo dejando su vida a 0**. Para ello, se puede realizar daño directo **alineando calaveras**, o bien acumulando el maná correspondiente para hacer uso de las distintas **magias** que posee el personaje. Dichas magias pueden provocar un daño directo al enemigo, pero aquí es donde entra en juego la estrategia, ya que existen gran variedad de tipos de magias con otros efectos, como puede ser la curación, el envenenamiento, la modificación del tablero o la alteración del maná propio o del oponente.

Gracias a la incorporación de las magias, el **sistema de combate** se vuelve más **estratégico y dinámico**, ya que cada jugador tendrá ciertas necesidades y preferencias a la hora de realizar una determinada línea u otra dependiendo del maná que se necesite para la magia que se quiere realizar en un momento determinado.

Adicionalmente en las batallas se pueden alinear casillas de experiencia y de oro, que permiten al usuario obtener un incremento adicional de dichos valores al final de la batalla.

Ilustración 2-14: Combate en Puzzle Quest

Otras características menos relevantes que posee *Puzzle Quest* son:

- Se puede **comprar equipo** para el personaje con el oro obtenido en las batallas. El equipo generalmente beneficiará en una mejora de las estadísticas del personaje.
- Se pueden **capturar monturas** en el juego, derrotando a ciertos personajes especiales, que permite moverse por el mapa a mayor velocidad y realizar un ataque especial.
- Se puede **desarrollar la ciudad** principal y capturar otras ciudades, que se establecen como bases para realizar diferentes acciones como comprar equipo para los combates o aprender nuevas magias.

2.4 Inteligencia Artificial en los videojuegos

La **Inteligencia Artificial** (IA) tiene como objetivo la aplicación de técnicas automáticas que permitan **resolver problemas para los cuales sería preciso el razonamiento humano**. La disciplina que lleva ese nombre se viene centrando históricamente en la elaboración de sistemas complejos que tienen en común un fuerte componente algorítmico y, normalmente, un elevado coste computacional.

La mayoría de los videojuegos actuales requieren, en mayor o menor medida, el uso de la Inteligencia Artificial, pudiendo utilizarse gran variedad de técnicas para el desarrollo de la misma. La Inteligencia Artificial se trata de un **factor clave a la hora de desarrollar un videojuego** ya que influye directamente en la jugabilidad del mismo, pudiendo convertir un título brillante en el resto de apartados, en un título molesto y tedioso para jugar en el caso de que la IA no esté a la altura.

Existen gran cantidad de técnicas de Inteligencia Artificial de las cuales muchas se han usado en algún videojuego para desarrollar su IA. El videojuego *Puzzle Quest*, en el cual se inspira este proyecto, no es una excepción y hace uso de técnicas de Inteligencia Artificial para simular el comportamiento del oponente en las batallas. A continuación se mencionarán las técnicas más relevantes que podrían encajar para dotar de IA al oponente en el juego a desarrollar.

2.4.1 Técnicas de Inteligencia Artificial

Existe una gran variedad de técnicas para el desarrollo de Inteligencia Artificial, de las cuales muchas de ellas han sido implementadas y usadas en la industria del videojuego para el desarrollo de la IA. En este punto no se pretende abarcar todas las técnicas existentes, sino ofrecer una breve descripción de las técnicas más relevantes que se podrían utilizar para implementar la IA del oponente en el juego que se pretende desarrollar en este proyecto.

- **Árboles y reglas de decisión**: Los árboles de decisión son una técnica de *aprendizaje supervisado*[19] muy utilizada en la que el conocimiento obtenido en el proceso de aprendizaje se representa mediante un árbol en el cual **cada nodo interior contiene una pregunta sobre un atributo** concreto (con un hijo por cada posible respuesta) **y cada hoja del árbol se refiere a una decisión** (una clasificación). Un árbol de decisión puede usarse para clasificar un caso comenzando desde su raíz y siguiendo el camino determinado por las respuestas a las preguntas de los nodos internos hasta que encontremos una hoja del árbol. Un ejemplo de árbol para decidir si se puede jugar al tenis sería:

[19] El **aprendizaje supervisado** es una técnica para deducir una función a partir de datos de entrenamiento. Los datos de entrenamiento están formado por una serie de atributos de entrada y sus correspondientes resultados deseados.

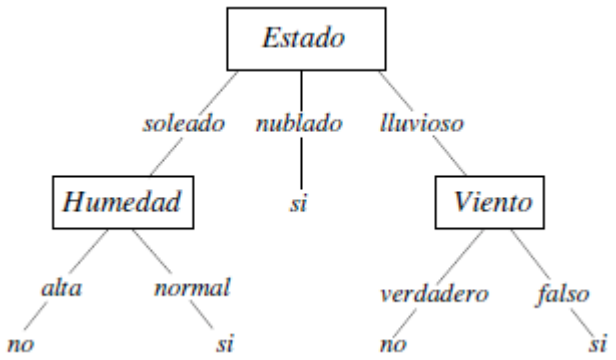

Ilustración 2-15: Árbol de decisión para jugar al tenis

Cualquier árbol de decisión se puede convertir a un conjunto de **reglas de decisión de tipo "si-entonces"** de forma que de cada camino desde la raíz del árbol hasta un nodo hoja se deriva una regla cuyo antecedente es una conjunción de literales relativos a los valores de los atributos situados en los nodos internos del árbol y cuyo consecuente es la decisión a la que hace referencia la hoja del árbol (la clasificación realizada). Para el ejemplo anterior quedaría de la siguiente forma:

SI *Estado* = *nublado* ENTONCES si

SI *Estado* = *soleado* Y *Humedad* = *alta* ENTONCES no

SI *Estado* = *soleado* Y *Humedad* = *normal* ENTONCES si

SI *Estado* = *lluvioso* Y *Viento* = *verdadero* ENTONCES no

SI *Estado* = *lluvioso* Y *Viento* = *falso* ENTONCES si

Las reglas se interpretan de una forma jerárquica siendo la más prioritaria la primera, y la menor la última. De esta forma un ejemplo será clasificado por la regla i-ésima si éste no cumple las condiciones establecidas por las i-1 reglas anteriores, es decir, cuando las anteriores reglas no han sido satisfechas.

Tanto para los árboles como para las reglas de decisión, la mayor parte de los algoritmos que las generan producen un número de condiciones suficientes como para que todos los ejemplos del fichero de entrenamiento sean cubiertos.

Esta técnica encajaría para el desarrollo de la IA del proyecto ya que permitiría, tomando los valores del estado del personaje y del tablero como atributos, obtener una salida sobre el siguiente movimiento a realizar por parte del jugador automático. Hay que destacar que ambas técnicas, tanto árboles como reglas de decisión, se adaptarían perfectamente para llevar a cabo la codificación de esta técnica.

- **Algoritmos genéticos**: Los algoritmos genéticos son métodos adaptativos, basados en la teoría de la evolución de Darwin, que pueden usarse para resolver problemas de búsqueda y optimización.

  Los algoritmos genéticos **trabajan con individuos**, que se corresponden a una solución concreta factible al problema dado. Los individuos están compuestos por un conjunto de cromosomas, que a su vez se descomponen en un conjunto de genes. La estructura genética del individuo **codifica el estado en el espacio de búsqueda** y dicho espacio es la combinación de todos los posibles valores que puede tomar esa estructura.

  EL funcionamiento de la técnica consiste en generar de forma aleatoria una **población inicial de individuos que va evolucionando mediante técnicas que imitan a la naturaleza y su aleatoriedad, como la selección natural, el cruzamiento o la mutación**. Para ello, a cada individuo se le asigna un valor o puntuación, relacionado con el grado de bondad de dicha solución. Cuanto mayor sea la adaptación de un individuo al problema, mayor será la probabilidad de que el mismo sea seleccionado para reproducirse, cruzando su material genético con otro individuo seleccionado de igual forma. Este cruce producirá nuevos individuos descendientes de los anteriores los cuales comparten algunas de las características de sus padres. Adicionalmente se puede producir una mutación de los nuevos individuos con la finalidad de favorecer la diversidad y explorar la búsqueda de nuevas soluciones.

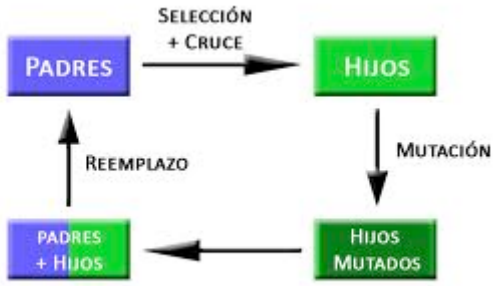

Ilustración 2-16: Fases de los algoritmos genéticos

La técnica produce una **sucesión de generaciones de individuos** descendientes de aquellos que estaban mejor adaptados en la generación anterior. De esta forma van mejorando los individuos en las sucesivas generaciones hasta llegar a un punto en el que no se produce dicha mejora, que es considerado como un estancamiento de la población; o bien en el que **se encuentra algún individuo que satisface** una serie de condiciones preestablecidas que componen **la meta**, finalizando la búsqueda.

Lo aleatorio de esta técnica hace que consiga un mejor rendimiento que otras cuando se trata de **buscar soluciones próximas a la óptima** (subóptimas), pero cada ejecución del algoritmo puede arrojar un resultado diferente. A pesar de ello, se trata de una técnica computacionalmente costosa, por lo que no suele emplearse en tiempo real sino, por ejemplo, en casos de optimización como pueda ser la asignación de los mejores parámetros para un personaje del juego o la codificación de un programa que cumpla una serie de restricciones.

El enfoque de esta técnica en el desarrollo de la IA del proyecto se basaría en establecer los individuos como un jugador automático, codificándolos como un conjunto de reglas que determinarían los movimientos a realizar de acuerdo al estado de la partida. Cada individuo simularía una serie de combates en el juego y se establecerían puntuaciones de acuerdo a sus resultados, estableciendo los correspondientes mecanismos de selección, cruce y mutación para obtener nuevas generaciones de individuos hasta llegar a aquel que presente un comportamiento que cumpla las expectativas deseadas.

- **Redes de neuronas**: Las redes de neuronas son una técnica de aprendizaje automático **inspirada en el funcionamiento del sistema nervioso de los animales**. Se trata de un sistema de capas de neuronas en las que las neuronas de cada capa están interconectadas con las neuronas de las capas adyacentes, permitiendo generar para cada estímulo de entrada su correspondiente salida.

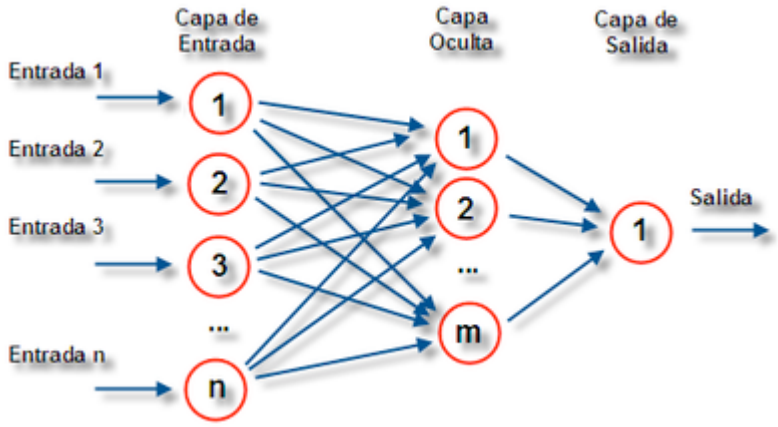

Ilustración 2-17: Estructura de una red de neuronas

El objetivo de las redes de neuronas es el de aprender complejas funciones no lineales que relacionan un conjunto de variables de entradas con una serie de variables de salida. Esto se consigue mediante el **ajuste de los pesos que tienen cada una de las conexiones que existen entre las diferentes neuronas**. Los pesos son modificados a lo largo de un **proceso de entrenamiento** de aprendizaje supervisado

en el cual se van introduciendo en la red las diferentes entradas del conjunto de entrenamiento y se compara la salida deseada con la que establece la red, realizando una modificación de los pesos en caso necesario para ajustar ambas salidas.

Uno de los inconvenientes a la hora de elegir esta técnica es que **se trata de un sistema de *'caja negra'***, esto es, que la complicada arquitectura que muchas veces es necesaria para resolver ciertos problemas **impide conocer la importancia que**, una vez producido el entrenamiento, **presenta cada variable** en la respuesta final del sistema. De esta forma esta técnica presenta un grave problema si el objetivo no es únicamente obtener una respuesta sino conocer también todo el proceso de razonamiento seguido.

No obstante, esta técnica podría ser usada para el desarrollo de la IA del proyecto, ya que únicamente es necesario decidir sobre el movimiento a realizar en cada momento y no es necesario conocer el porqué de dicha decisión. En este caso, cada valor del estado del personaje y del tablero sería una entrada de la red de neuronas, y la salida de la misma sería el siguiente movimiento a realizar por parte del jugador automático.

2.4.2 Técnica utilizada: árboles y reglas de decisión

Cualquiera de las técnicas anteriormente comentadas podría utilizarse para el desarrollo de la IA del proyecto obteniendo notables resultados. No obstante, finalmente la técnica que se empleará en el desarrollo del proyecto es la de *'árboles y reglas de decisión'* por diversas razones que se explicarán a continuación:

- **Sencilla representación del estado**: La técnica de árboles y reglas de decisión únicamente necesita ir guardando para cada acción realizada en el modo combate los valores de los atributos de los jugadores en ese momento, el estado del tablero y la acción realizada. Estos valores se obtienen directamente del módulo funcional del juego, sin que sea necesario a priori realizar cálculos para la obtención de los mismos.
- **Facilidad de aplicación de la técnica**: Una vez guardados los datos que representan el estado del combate para cada acción realizada únicamente es necesario usar uno de los muchos programas que existen de *Aprendizaje automático*[20] para obtener los resultados de la técnica. En este aspecto **salen como claros perdedores los algoritmos genéticos**, que necesitan montar un sistema de codificación de individuos y de selección, cruce y mutación de los mismos, resultando muy costosa la aplicación de dicha técnica al desarrollo del proyecto.

[20] El **Aprendizaje Automático** es una rama de la Inteligencia Artificial cuyo objetivo es desarrollar técnicas que permitan a las computadoras *aprender*. De forma más concreta, se trata de crear programas capaces de generalizar comportamientos a partir de una información no estructurada suministrada en forma de ejemplos.

- **Se trata de un módulo fácilmente adaptable**: Los datos guardados para cada acción realizada en el modo combate permiten obtener un árbol y/o unas reglas de decisión que se pueden codificar en *C#* por medio de sentencias de tipo "si-entonces". De esta forma los resultados obtenidos en el proceso de aprendizaje quedan simplificados a un fragmento de código en el que se encuentran dichas sentencias y que permite obtener, a partir de los atributos que definen el estado del combate, una salida con la acción a realizar. En este sentido tienen mayores **dificultades las redes de neuronas** ya que la solución que genera dicha técnica en la mayor parte de los casos produce fórmulas más complejas y difícilmente interpretables, por lo que sería más complejo codificar dichas fórmulas en el lenguaje *C#*.
- **Permite obtener resultados satisfactorios**: Este aspecto se ha comprobado a posteriori del desarrollo de la técnica, certificando que los resultados que se han obtenido han generado un módulo de IA cuya salida produce acciones coherentes y aceptables.

2.4.3 Herramienta utilizada: Weka

Una vez elegida la técnica a utilizar se procede a realizar una exploración de las posibles herramientas comerciales que permiten llevarla a cabo, adicionalmente también se valora la posibilidad de **codificar propiamente los algoritmos** para generar los árboles y reglas de decisión. Esta segunda opción **queda totalmente descartada** ya que incrementaría el esfuerzo innecesariamente centrándolo en la codificación de los algoritmos relegando a segundo lugar la investigación de las posibles soluciones y la comparación de diferentes algoritmos y pruebas, que es el aspecto que realmente se considera importante.

En lo que respecta a las herramientas comerciales directamente se pensó en la **herramienta *Weka***[21] ya que anteriormente se había trabajado con ella, habiendo siempre obtenido resultados satisfactorios. Además se trata de una herramienta totalmente gratuita, que ofrece una gran cantidad de algoritmos y opciones de tratamiento de atributos y filtros, todo ello con una interfaz realmente sencilla y fácil de interpretar.

[21] Página principal de Weka: http://www.cs.waikato.ac.nz/~ml/weka/

3 Objetivos

El objetivo principal de este proyecto es la **creación de un videojuego para PC mediante la tecnología *XNA* de *Microsoft*, inspirado en el actual videojuego *Puzzle Quest***. De esta forma se pretende culminar la carrera de Ingeniería en Informática con especialización en Inteligencia Artificial con un PFC que supone una motivación personal, permitiendo de este modo adquirir los conocimientos necesarios para introducirse en el mundo del desarrollo de videojuegos, y realizando un aporte a dicha comunidad de forma que se pueda continuar el desarrollo del mismo posteriormente.

Existen una serie de **objetivos secundarios que se intentarán cumplir en su mayoría** a lo largo del desarrollo:

1. **El videojuego desarrollado debe tener una jugabilidad aceptable**: La jugabilidad es uno de los factores más importantes a la hora de elaborar un videojuego ya que es lo que diferenciará un juego adictivo de un juego aburrido y sin brillo propio. Por este motivo se intentará vigilar aspectos como los gráficos, el feedback y el sistema de juego.
2. **Se permitirá elegir entre distintos tipos de personajes**: Esta elección tendrá lugar al comienzo de la partida e influirá en las características del personaje como la vida, el maná inicial y el conjunto de magias a usar. Así mismo, a medida que se vaya progresando en el juego y ganando combates se irá obteniendo experiencia que permitirá subir de nivel, mejorando los atributos del personaje y permitiendo aprender nuevas magias.
3. **Existirá un mapa principal para buscar misiones y combates**: El objetivo es que el jugador pueda mover su personaje por el mapa buscando nuevas misiones y aceptando nuevos combates. Una vez aceptado un combate se establecerá un juego de tipo tablero con el cual se lleva a cabo el desarrollo del mismo.
4. **El modo combate mezclará rol con un tablero estilo *Bejeweled***: Este es el estilo que precisamente combina a la perfección el juego *Puzzle Quest* en el cual se inspira el proyecto. Para ello se basa en el juego de tablero de *Bejeweled* pero destinando su uso para simular un combate 2 jugadores por turnos, de modo que las líneas generadas permitan obtener maná que será necesario para utilizar las distintas magias que tiene cada uno de los personajes.
5. **Se generará un log para guardar el desarrollo de los combates**: Se trata de un requisito imprescindible si se quiere aplicar alguna técnica de Inteligencia Artificial como es el caso. De este modo se llevará un registro que permitirá obtener información de utilidad sobre el comportamiento de los jugadores en el combate, pudiendo, mediante técnicas de IA, simular dicho comportamiento.

6. **Se desarrollará un jugador automático con técnicas de IA**: Para conseguir este objetivo es necesario implementar un método en juego que permita recibir el estado actual del combate mediante parámetros y que devolverá un código que se interpretará como la acción que el jugador automático pretende realizar. Otro factor de este punto es que el jugador automático a desarrollar debe obtenerse a partir de técnicas de Inteligencia Artificial.
7. **Se incorporará un modo para 2 jugadores**: En este caso no se trata de un modo de juego completo sino que el objetivo es que se trate de un combate rápido entre ambos jugadores, permitiendo elegir a cada uno el personaje con el que quiere jugar. También se puede ampliar a un modo de jugador humano contra jugador automático para realizar combates rápidos cuando solo hay una persona jugando; e incluso sería interesante un modo de 2 jugadores automáticos, para observar cómo se desenvuelven los distintos jugadores creados con las distintas técnicas de IA que se hayan podido utilizar.
8. **Debe ser todo lo sencilla posible la posibilidad de continuación del desarrollo del videojuego por terceros**: *XNA* es una plataforma de desarrollo de código abierto en la que existen una gran cantidad de desarrolladores. Debido a que al juego desarrollado seguramente le queden pendientes muchas líneas futuras sería muy interesante que otros desarrolladores pudieran continuar y ampliar el videojuego desarrollado en este proyecto.
9. **El idioma de desarrollo del videojuego debe ser el inglés**: De esta forma el código generado está en sintonía con el nombre de todas las clases y objetos del propio *XNA*, que están en inglés**.** Sin embargo, los comentarios se han realizado en español, ya que permitirán que el proceso de desarrollo sea más sencillo al tener mayor conocimiento de dicho lenguaje. No obstante, no se descarta que tras la realización de este proyecto se traduzcan también los comentarios para permitir la continuación con el desarrollo a un conjunto más amplio de la comunidad de desarrolladores**.**

4 Desarrollo

4.1 Introducción

El videojuego *Magic Quest*, fruto de este PFC, se ha desarrollado íntegramente en el mismo. Para ello, únicamente se ha tomado como origen la plantilla inicial que genera *XNA* automáticamente cuando se crea un proyecto de desarrollo de un videojuego para PC, y a partir de la misma se han ido introduciendo los diferentes elementos del juego y generando el código del mismo hasta llegar al producto resultante final. Por lo tanto, no se ha utilizado como base ni se ha introducido fragmentos de código de ningún otro juego desarrollado con XNA, aunque sí se han utilizado de referencia para comprender la estructura y el funcionamiento de un videojuego desarrollado con *XNA*.

En los siguientes apartados **se realizará una descripción del producto final desarrollado, detallando los diferentes módulos del mismo**, y obviando los pasos y posibles problemas que se han ido produciendo durante el desarrollo de los mismos. Se hará hincapié en el módulo de Inteligencia Artificial, exponiendo un estudio sobre los distintos experimentos realizados hasta llegar al resultado final obtenido en este proyecto.

4.2 Análisis

La fase de análisis permite identificar las necesidades del producto a desarrollar mediante la obtención de los requisitos funcionales y no funcionales, ayudando al desarrollador a comprender la naturaleza del producto a construir, así como detectar las funcionalidades requeridas, el comportamiento, el rendimiento y las interconexiones del mismo. En el caso de este proyecto se realizará un análisis no exhaustivo, basado en la **descripción de los casos de uso**.

4.2.1 Actores

La funcionalidad principal de *Magic Quest* es la del entretenimiento jugando partidas al mismo, pero adicionalmente está ideado para aportar otras funcionalidades que permitan cubrir otros objetivos y necesidades que puedan tener los usuarios del producto. Por lo tanto de acuerdo al tratamiento que se realice con *Magic Quest* se han detectado **3 tipos de roles** que pueden hacer uso del mismo: **Jugador, Investigador IA y Desarrollador**. Se trata de roles que pueden llegar a ser complementarios, ya que un mismo usuario, dependiendo de la funcionalidad que esté desarrollando con el videojuego, puede adoptar distintos roles dentro del mismo. A continuación se exponen los distintos roles que pueden adoptar los usuarios de *Magic Quest*.

- **Jugador**: Se trata del **rol más común** a la hora de utilizar un videojuego. Cualquier persona puede desempeñar este rol ya que **no se requiere de ninguna capacidad ni conocimiento concreto**. Las personas que desempeñen este rol, si poseen las características necesarias, pueden a su vez desempeñar otros roles de los que se expondrán a continuación.
- **Investigador IA**: Este rol será desempeñado por las personas que deseen **experimentar con el comportamiento del jugador automático del juego**. Para favorecer esta labor, *Magic Quest* proporciona un log del desarrollo de los combates, que será de utilidad a este tipo de usuarios para interpretar y comprender comportamientos a la hora de jugar. Además, el código que establece el comportamiento de la IA del jugador automático está encapsulado en un único método de una clase independiente que recibe como parámetros el estado actual del combate y devuelve como salida el movimiento a realizar; este código puede ser modificado por los usuarios pertenecientes a este rol para alterar el comportamiento de la IA. Las capacidades que se consideran necesarias para poder desempeñar este rol son el **conocimiento de técnicas de aprendizaje automático y de Inteligencia Artificial**, y en cierto modo también es necesario tener algunos conocimientos de programación si se quiere trabajar con el comportamiento de la IA. Las personas que desempeñen este rol generalmente también asumirán el rol de Jugador ya que será necesario que jueguen partidas para comprobar los avances de la IA desarrollada, aunque no necesariamente porque puede que únicamente les interese el estudio de los comportamientos a la hora de jugar sin que sea él mismo el que juegue dichas partidas.
- **Desarrollador**: Se trata de un rol un tanto especial ya que, al contrario que los anteriores, el rol de desarrollador no hace realmente un uso del juego, sino que asumirán este rol todas las personas que centren sus esfuerzos en **modificar el código del videojuego** para realizar cambios en el mismo **y aportar nuevas funcionalidades**. El código de *Magic Quest* está estructurado intentando facilitar esta labor ya que, por ejemplo, se ha intentado independizar en la medida de lo posible las características y magias de los personajes, de forma que en un futuro nuevos desarrolladores puedan introducir fácilmente nuevos personajes, modificar sus características y/o modificar sus magias correspondientes. Las capacidades para desempeñar este rol son **conocimientos del lenguaje de programación** del videojuego **y conocimientos de la codificación** actual del mismo. Las personas que desempeñen este rol no tienen necesariamente porque asumir los roles anteriores aunque, con bastante seguridad, adoptarán también el rol de Jugador, ya que será necesario usar el sistema y jugar ciertas partidas para probar los cambios introducidos.

4.2.2 Casos de uso

Una vez establecidos los diferentes roles que harán uso del sistema se determinan, teniendo en cuenta las funcionalidades esperadas para el producto, el siguiente diagrama de casos de uso:

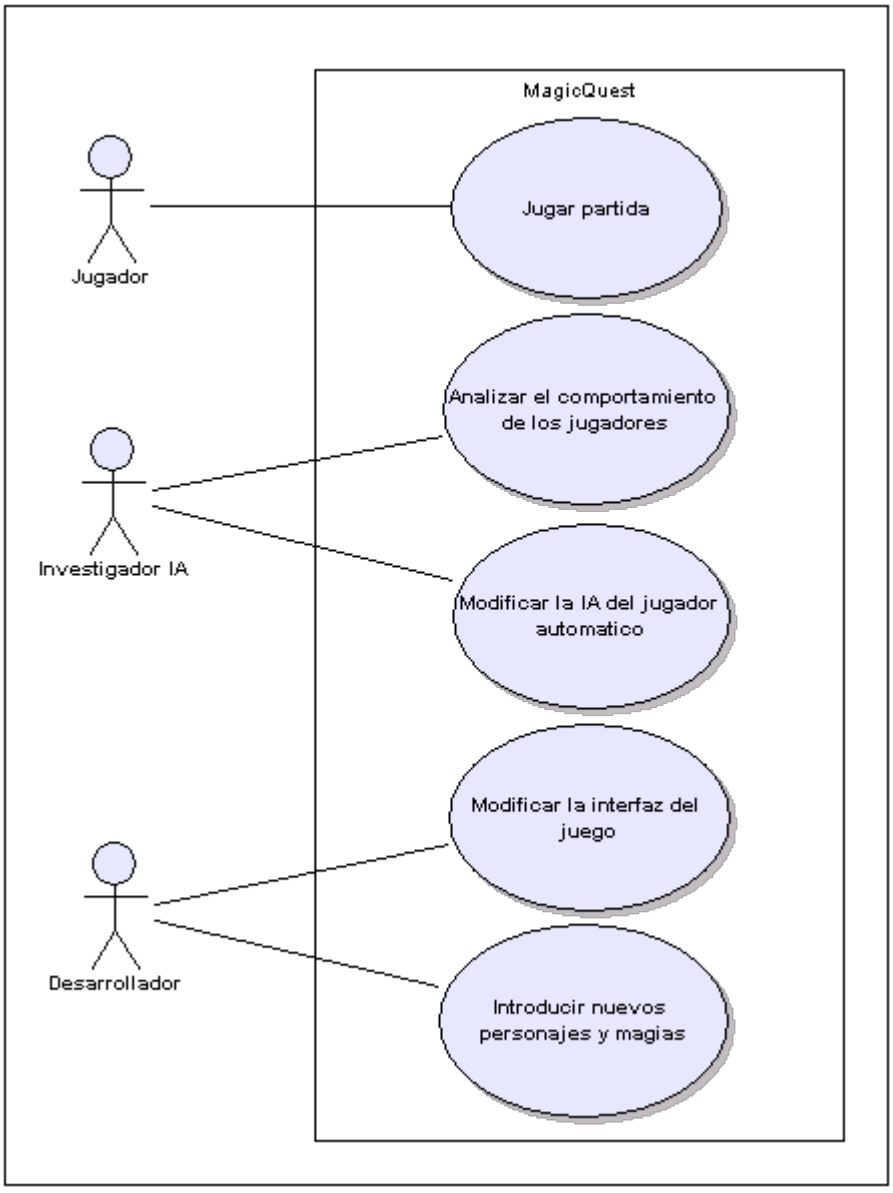

Ilustración 4-1: Diagrama de casos de uso

El diagrama de casos de uso refleja 3 actores que se corresponden con los roles identificados en el punto anterior: Jugador, Investigador IA y Desarrollador. Cada uno de ellos está asociado a uno o varios casos de uso de acuerdo a su adecuación dado sus capacidades y conocimientos. A continuación se describen los distintos casos de uso detectados.

- **Jugar partida**: Se trata de la funcionalidad principal que debe tener todo videojuego. *Magic Quest* proporciona al usuario **varias modalidades de juego: *1 Jugador*, *2 Jugadores*, *Jugador Vs IA* y *Automático***. Las partidas en modo *1 Jugador* tienen un sistema de juego clásico basado en el juego *Bejeweled*, que consta únicamente de un tablero y cuyo objetivo es el de generar el mayor número de líneas posibles antes de que se agote el tiempo establecido. El resto de modos comparten el sistema principal de juego, que también está basado en el juego de tablero de *Bejeweled* pero destinando su uso para simular un combate 2 jugadores por turnos cuyo objetivo es reducir la vida del jugador contrario a 0, bien realizando líneas que produzcan al enemigo un daño directo, o bien utilizando alguna magia que penalice al enemigo o reduzca su vida. Para poder utilizar las diferentes magias será necesario obtener sus correspondientes cantidades de los distintos tipos de maná requeridas, y estas se consiguen realizando líneas de casillas de diferentes tipos.

- **Analizar el comportamiento de los jugadores**: Magic Quest proporciona un log del desarrollo de los combates, que será de utilidad a los usuarios que adquieran el rol de Inverstigador IA para poder interpretar y comprender comportamientos a la hora de jugar. En concreto **existen 3 tipos de logs que guardan**, con distintos matices, **el estado del combate y la acción llevada a cabo en cada turno**. Dichos logs podrán ser analizados con técnicas de aprendizaje automático para obtener conocimiento a partir de los mismos. No obstante, usuarios más técnicos podrán adaptar el código del juego para obtener la información que consideren oportuna, modificando los datos que se guardan en los logs o bien estableciendo algún otro sistema para obtener los datos necesarios.

- **Modificar la IA del jugador automático**: El código que establece el **comportamiento de la IA** del jugador automático está **encapsulado en un único método de una clase independiente**. El módulo de juego de *Magic Quest* se encarga de llamar periódicamente a dicho método en cada turno de un jugador automático, enviando como parámetros el estado actual del combate y esperando como devolución el movimiento a realizar por dicho jugador. Gracias a esta independencia, los usuarios que posean las capacidades oportunas podrán modificar dicho código para alterar el comportamiento de la IA. Esta funcionalidad está relativamente ligada a la anterior, ya que ofrece un gran potencial analizar el comportamiento de los jugadores con técnicas como el aprendizaje automático, para obtener una serie de reglas que finalmente serán codificadas en el método que controla la IA del jugador automático para que adquiera dicho comportamiento.

- **Modificar la interfaz del juego**: La interfaz gráfica está formada por texturas 2D, también llamadas ***sprites***. Los desarrolladores con los conocimientos de modelado gráfico necesario pueden crear sus propios *sprites* y reemplazarlos por los que actualmente aparecen en el juego. Además, **los datos de los distintos elementos correspondientes a sus dimensiones y localización en la pantalla de juego, están centralizados** en una única clase estática, que será consultada por el resto de clases a la hora de crear los elementos. La ventaja que aporta esta clase es que centraliza dichos atributos y facilita la búsqueda y modificación de las características de los elementos que se dibujan por pantalla de modo que únicamente es necesario modificar esta clase para aplicar todos los cambios que se deseen realizar en la interfaz del juego.

- **Introducir nuevos personajes y magias**: Se ha estructurado el código correspondiente a las **características y magias** de los personajes de forma que **esté lo más independiente posible** respecto al resto del código del juego. De esta forma se pretende minimizar los esfuerzos para futuros desarrolladores a la hora de modificar los distintos personajes del juego e incluso crear nuevos personajes. Respecto a las magias, también se pueden modificar o añadir nuevas aunque el código de las mismas se encuentra más acoplado al núcleo y será necesario un mayor conocimiento de la codificación del juego para poder realizar dicha tarea. Por último cabe mencionar que no se cierra las puertas a que desarrolladores más avanzados puedan cambiar cualquier otro factor del juego, pudiendo ser modificaciones en el tablero, en el propio sistema de combates e incluso en la propia estructura y desarrollo del juego.

4.3 Diseño conceptual

El diseño conceptual recoge las decisiones tomadas en relación al diseño del sistema, tales como la **arquitectura del sistema**, la definición de los distintos **componentes** de los que se forma y la descripción de los mismos. No entra en detalles de implementación, los cuales serán descritos posteriormente en el diseño detallado.

4.3.1 Arquitectura de la aplicación

La arquitectura de la aplicación permite obtener un diseño a alto nivel del sistema, identificando y definiendo los **módulos de los que consta**, así como las **relaciones que existen entre los mismos**. El diseño realizado en este proyecto no está basado en ningún modelo de arquitectura específico existente, sino que se ha realizado un diseño específico para el videojuego desarrollado. Las prioridades a la hora de construir el diseño han sido maximizar la cohesión entre los componentes de cada uno de los módulos y minimizar el grado de dependencia entre los distintos módulos con el fin de favorecer la reusabilidad.

El diagrama de componentes desarrollado para *Magic Quest* es el siguiente:

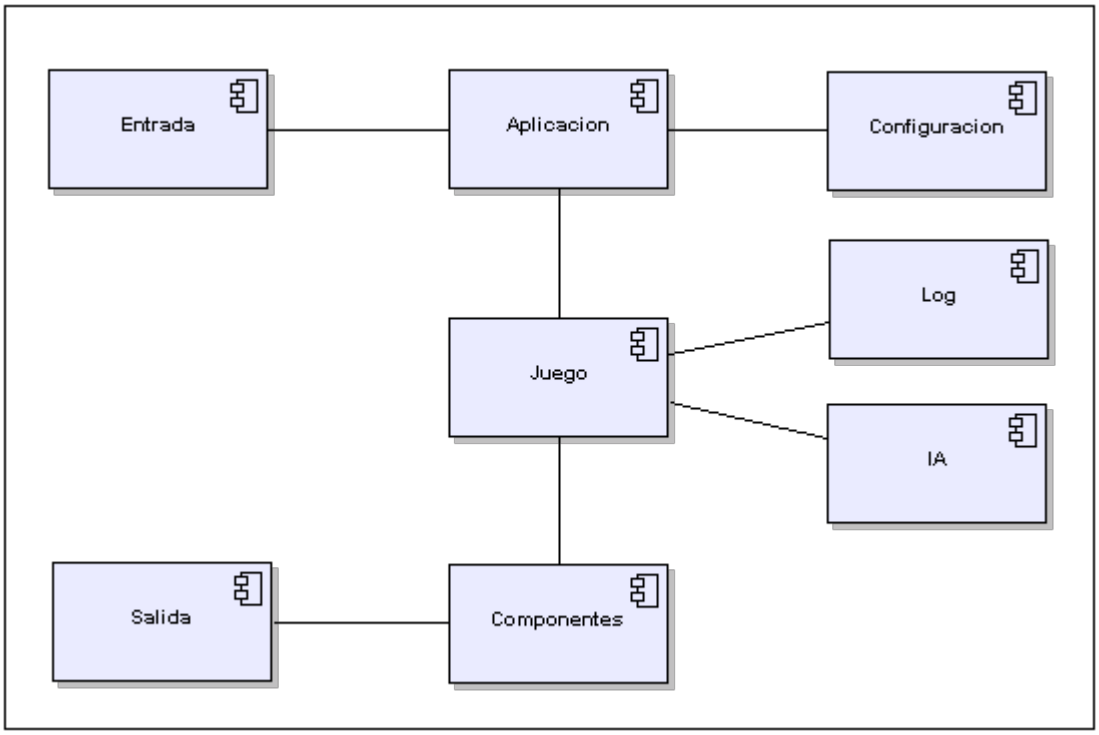

Ilustración 4-2: Diagrama de componentes

Este diagrama de componentes muestra los distintos módulos de los que se compone *Magic Quest*, así como las relaciones que existen entre los mismos. A continuación se expondrá brevemente el papel de cada uno de ellos en el conjunto, siendo definidos con mayor detalle en posteriores apartados.

- **Módulo de aplicación**: Se trata del módulo central encargado de llevar a cabo la **ejecución del juego** y controlar los elementos del núcleo del mismo. Establece el control de las distintas pantallas que componen al juego y la transición entre las mismas de acuerdo las distintas circunstancias y acciones que se realicen en la partida. Para realizar satisfactoriamente dicha función, este módulo guarda una serie de estados que permiten controlar la situación del juego en cada momento.

- **Módulo de juego**: Se trata del **controlador del sistema de combate** del juego. Su elemento principal es el tablero y está encargado de establecer su comportamiento y su correcto funcionamiento. Adicionalmente para el modo *1 jugador* también establece el marcador y el temporizador necesarios. Para el resto de modos en lugar de dichos elementos establece los dos jugadores que llevarán a cabo el combate, encargándose de gestionar sus correspondientes acciones y el sistema de turnos. Este módulo es el encargado de administrar al módulo log para registrar la partida y al módulo IA, enviándole cuando sea su turno el estado actual del combate y recibiendo y realizando la acción elegida.

- **Módulo de componentes**: El objetivo de este módulo es **encapsular los datos de los diferentes objetos que se muestran por pantalla** de forma que todos tengan ciertas características y funcionalidades comunes, facilitando el tratamiento y uso de los mismos. Existen diferentes tipos de componentes según la naturaleza de lo que se desee encapsular. De este modo se ha decidido crear un componente para las imágenes y botones, otro para las magias, otro para las casillas y un último para los textos. Una de las características que se aporta a dichos componentes es, por ejemplo, la de permitir redimensionar los distintos elementos en el caso de que se redimensione el tamaño de la ventana de juego; además se dota de homogeneidad a la hora de mostrar por pantalla dichos elementos y se tiene en cuenta si dicho elemento está redimensionado a la hora de hacerlo. Con todo ello, este módulo permite al módulo de juego tratar a todos sus componentes de un modo común facilitándole el tratamiento de los mismos.

- **Módulo de configuración**: Se trata de un módulo que **centraliza los datos de configuración del juego** y de los distintos elementos que lo componen. La utilidad de este módulo es la de facilitar la búsqueda y la modificación de las características de los elementos que se dibujan por pantalla de modo que únicamente es necesario modificar este módulo para aplicar todos los cambios que se deseen realizar. Además este módulo **independiza** en la medida de lo posible **las características y magias de los personajes**, de forma que será el módulo a modificar en caso de querer introducir nuevos personajes, modificar sus características y/o modificar sus magias correspondientes en un futuro.

- **Módulo de log**: Este módulo está encargado de generar un log para registrar el desarrollo de los combates a partir de la información recibida por parte del módulo de juego. En concreto este módulo **genera 3 tipos de logs que guardan**, con distintos matices, **el estado del combate** y la acción llevada a cabo en cada turno.

- **Módulo de IA**: Este módulo se encarga de establecer el comportamiento de la **IA del jugador automático** a partir de la información recibida por parte del módulo de juego y permite devolverle al mismo el movimiento a realizar por dicho jugador. El comportamiento de este módulo en el sistema desarrollado en este proyecto está basado en las reglas de decisión generadas por el algoritmo *JRip* del programa de aprendizaje automático *Weka* a partir de los datos obtenidos con el módulo de log descrito en el punto anterior. Este módulo está preparado para permitir al usuario modificar el comportamiento de la IA, siempre y cuando se respete e interprete correctamente la información recibida por parámetro respecto a la situación del combate, y se genere un movimiento a realizar permitido y que siga la codificación establecida para que pueda ser interpretado por el módulo de juego.

- **Módulo de entrada**: Se trata de un módulo que actúa como **manejador de la entrada por parte del usuario**, encapsulando los datos recibidos y aportando ciertas funcionalidades que permitan facilitar su tratamiento y uso. Actualmente dicho módulo solo consta del manejador de ratón, ya que de momento el juego solo está adaptado para funcionar en *PC*; pero en caso de que se quisieran realizar los ajustes necesarios para que funcionara en *Xbox 360*, en este aspecto únicamente sería necesario crear un nuevo manejador para el mando de dicha consola. Una de las funcionalidades más importantes que actualmente tiene el manejador de ratón es la posibilidad de comprobar si el puntero se encuentra situado en una zona de la pantalla cuya situación se recibe por parámetro.

- **Módulo de salida**: Módulo que actúa como **controlador del dispositivo gráfico de ordenador**, facilitando al resto de módulos una serie de funcionalidades que permitirán mostrar sus correspondientes elementos por pantalla. Se trata de una extensión del controlador que proporciona el propio XNA que amplia los tipos de llamadas utilizados para dibujar los elementos y además incorpora nuevas funcionalidades como el redimensionamiento de los distintos elementos en el caso de que se redimensione el tamaño de la ventana de juego.

4.3.2 Capas de la arquitectura

El diseño de la arquitectura se ha realizado, como ya se ha comentado, intentando maximizar la cohesión entre los componentes de cada uno de los módulos y minimizar el grado de dependencia entre los distintos módulos con el fin de favorecer la reusabilidad. Para ello se ha decidido establecer una **arquitectura con 3 capas**, cuyas características son las siguientes:

- **Controladores**: Se trata de la capa del sistema encargada de la entrada/salida del mismo. Actúa como manejador de la entrada por parte del usuario, basada principalmente en el uso del ratón; y como controlador del dispositivo gráfico del ordenador, permitiendo la visualización de los elementos por pantalla.
- **Núcleo**: Se trata de la capa del sistema cuyos componentes forman la estructura principal del juego, estableciendo las características y el comportamiento del mismo, y controlando su correcta ejecución. En concreto establece el control de las distintas pantallas de juego así como del sistema de combate.
- **Suplementos**: Se trata de la capa del sistema que contiene funcionalidades adicionales que se encuentran suficientemente desacopladas al mismo. De esta forma los elementos pertenecientes a esta capa pueden ser fácilmente reemplazados, como ocurre con el comportamiento del jugador automático.

La clasificación de los distintos módulos del diagrama de componentes respecto a la arquitectura de 3 capas utilizada es la siguiente:

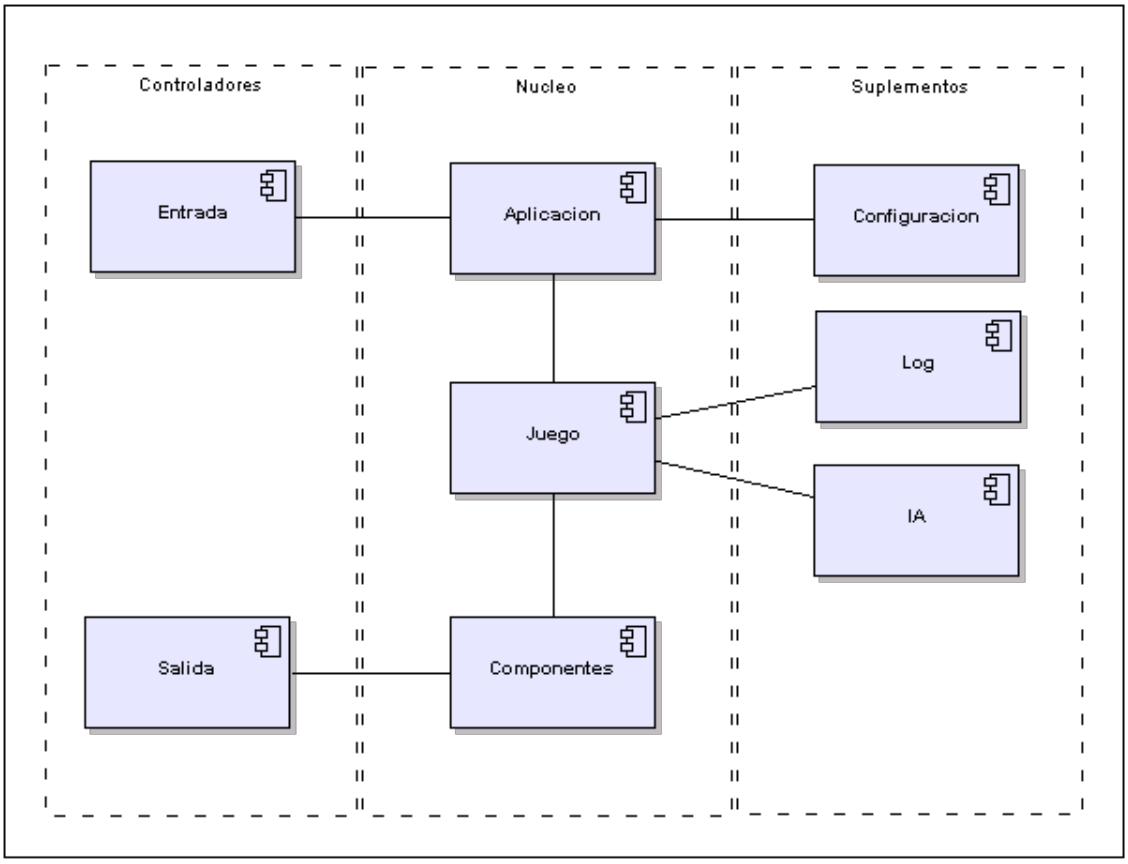

Ilustración 4-3: Capas de la arquitectura

La arquitectura representada en el modelo anterior agrupa a los módulos por funcionalidad, siendo la **capa *núcleo*** la formada por los módulos principales, actuando como **esqueleto del juego**. Las modificaciones en los elementos de esta capa actúan directamente en el sistema de juego, alterando las características del mismo y su funcionalidad.

Sin embargo, las otras dos capas no tienen un papel tan fundamental en el juego por lo que al realizar modificaciones en los elementos de dichas capas únicamente cambia el modo de trabajo de dichos elementos, pero no las características del juego o su funcionalidad. De este modo, la **capa *controladores*** podría ser modificada o reemplazada estableciendo otros mecanismos de entrada/salida al sistema; **y la capa *suplementos*** también **permitiría su modificación o reemplazo**, estableciendo nuevas configuraciones del juego, nuevas formas de guardar el log o nuevos comportamientos del jugador automático, pero sin que esto influya en el sistema de juego.

4.4 Diseño detallado

El diseño detallado realizado en este proyecto pretende establecer una definición más específica de cada uno de los componentes anteriormente descritos, estableciendo para cada uno de ellos las **clases que los componen**, y definiendo a su vez para cada uno de ellos sus **principales atributos y métodos**. En caso de considerarse oportuno también se definirán y explicarán algunos de los algoritmos utilizados en algún método específico.

La definición de las clases de los distintos componentes comenzará por la capa de *controladores* por tratarse de las clases más básicas e independientes del sistema. Posteriormente se procederá a especificar las clases de la capa del *núcleo*, comenzando de nuevo por su parte más básica que es el módulo componentes. Para finalizar las clases de los módulos de la capa *suplementos,* al estar basadas en el funcionamiento del *núcleo* del juego pero aportando funcionalidades adicionales al mismo, se especificarán en último lugar.

4.4.1 Módulo de entrada

El módulo de entrada pertenece a la capa de controladores y actúa como manejador de la entrada por parte del usuario, encapsulando los datos recibidos y aportando ciertas funcionalidades que permitan facilitar su tratamiento y uso. El diagrama de clases de dicho módulo es el siguiente:

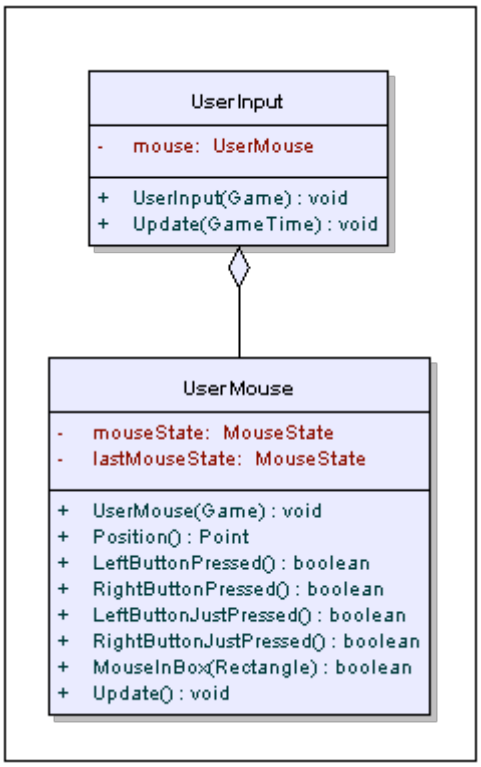

Ilustración 4-4: Módulo de entrada

Se trata de un módulo básico independiente del sistema desarrollado, que se podría reutilizar perfectamente en cualquier otro producto desarrollado sin apenas tener que realizar modificaciones. Como se puede observar, actualmente solo consta de 2 clases: *UserInput* y *UserMouse*. La primera de ellas, *UserInput*, actúa como centralizador de la entrada, permitiendo a cualquier sistema que quiera utilizar el módulo de entrada que únicamente tenga que declarar un objeto de dicha clase. La segunda, *UserMouse*, corresponde a una implementación concreta de un manejador de ratón, con todas las funcionalidades necesarias para obtener la entrada del mismo. En los siguientes párrafos se realizará una descripción más detallada de las mismas.

La clase ***UserInput*** actúa como centralizador de la entrada, facilitando a través de un único objeto el manejo de la entrada por parte de otros sistemas, que solo tendrán que declarar un objeto de la misma. Esta clase forma un agregado de todas las diferentes entradas que existan, aunque actualmente sólo existe la entrada del ratón ya que de momento el juego solo está adaptado para funcionar en *PC*. En un futuro se podría agregar a esta clase un manejador para el mando de la consola *Xbox 360*, encapsulando y centralizando la entrada de dicho periférico del mismo modo que se hace con el ratón.

Actualmente esta clase solo actúa como contenedor de las diferentes entradas del sistema, encargándose únicamente de actualizar el estado de las mismas a través del método *Update(GameTime)*. No se encarga de facilitar la transparencia en el uso de las diferentes entradas por parte de otros sistemas, ya que para cada tipo de entrada es necesario utilizar sus métodos específicos. Esto es así debido a que actualmente solo consta de la entrada de ratón, pero a medida que se vayan introduciendo nuevos tipos de entradas sería conveniente que la clase *UserInput* favoreciera su transparencia mediante la creación de métodos que a su vez sean los que se encarguen de consultar al tipo de entrada correspondiente.

La clase ***UserMouse*** actúa como manejador de la entrada del sistema proveniente del periférico ratón. Para ello se ayuda de la estructura *MouseState* que proporciona el propio *XNA* y que permite obtener información del estado de los distintos botones del ratón en cada momento. La clase *UserMouse* está diseñada para proporcionar información a partir del estado del ratón en un momento determinado con respecto al momento anterior, de esta forma contiene una serie de métodos de gran utilidad para su uso en el sistema como la posibilidad de detectar si alguno de los botones del ratón ha sido pulsado, o detectar la posición del puntero en la pantalla.

Una de las funcionalidades de mayor utilidad que proporciona esta clase es el método *MouseInBox(Rectangle)* que permite comprobar si el puntero se encuentra situado en la zona de la pantalla descrita por el rectángulo introducido por parámetro. De esta forma se puede detectar cuando se pulse algún botón en que objeto se ha realizado, e incluso asignar comportamientos especiales a los distintos objetos de la pantalla cuando el ratón se encuentre situado encima.

4.4.2 Módulo de salida

El módulo de salida pertenece a la capa de controladores y actúa como controlador del dispositivo gráfico de ordenador, facilitando al resto de módulos una serie de funcionalidades que permitirán mostrar sus correspondientes elementos por pantalla. El diagrama de clases de dicho módulo es el siguiente:

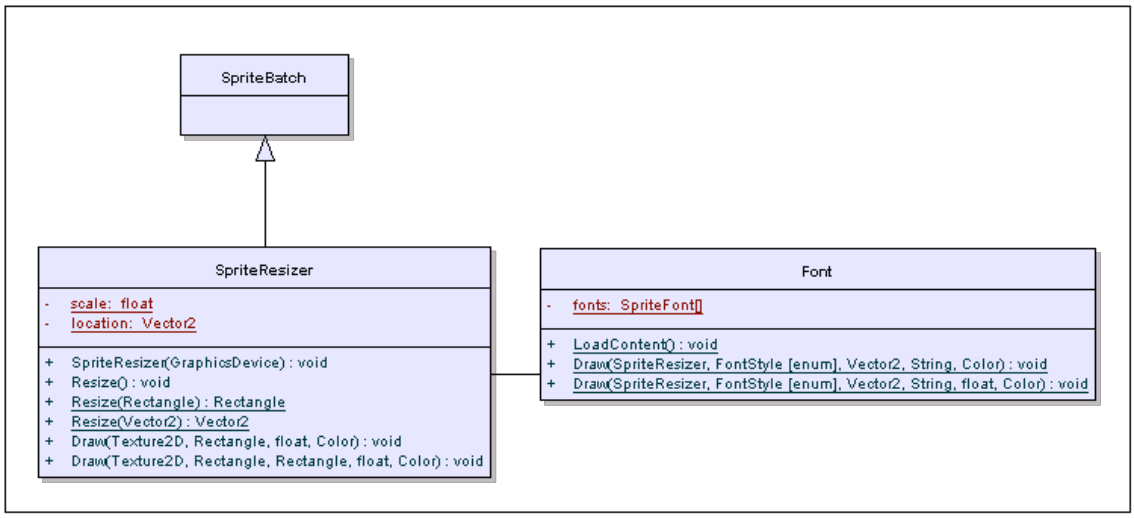

Ilustración 4-5: Módulo de salida

Se trata de un módulo algo más complejo que el de entrada, pero también básico e independiente del sistema desarrollado. Como el anterior también consta de 2 clases: *SpriteResizer* y *Font*, aunque en el diagrama se ha representado también *SpriteBatch* para dejar constancia de que la clase desarrollada *SpriteResizer* hereda de dicha clase propia de *XNA*. La funcionalidad de este módulo es la de permitir mostrar por pantalla las diferentes texturas 2D y texto del juego. A continuación se describe más específicamente cada una de las clases de las que consta este módulo.

La clase ***SpriteResizer*** es la encargada de dibujar las texturas 2D del juego, tal y como ya lo hace la propia clase que proporciona *XNA* llamada *SpriteBatch*, de la cual hereda, pero incorporando la funcionalidad de admitir la redimensión de los diferentes elementos a dibujar en el caso de que se cambie el tamaño de la ventana de juego. Para ello consta con dos atributos adicionales, *scale* y *location*, que guardan valores necesarios para ajustar las diferentes imágenes a la dimensión de la ventana de juego y centrarlas con respecto a la misma. Los valores de estos atributos se modifican mediante el método *Resize()*, llamado cada vez que se modifica el tamaño de la ventana de juego.

En lo que respecta al redimensionamiento esta clase también contiene los métodos estáticos *Resize(Rectangle): Rectangle* y *Resize(Vector2): Vector2* que tienen como finalidad permitir a otras clases obtener las nuevas dimensiones de sus correspondientes objetos en el caso de que se modifique el tamaño de la ventana de juego.

Para finalizar los métodos *Draw()* realizan la funcionalidad principal de la clase, ya que son los que permiten mostrar los elementos gráficos por pantalla. Su funcionamiento se basa en el propio método *Draw()* de la clase *SpriteBatch* de la cual hereda, pero con la particularidad de introducir el factor de redimensionamiento de los elementos. De esta forma no solo permite mostrar los elementos por pantalla sino que además permite adaptar su posición y dimensiones en función del tamaño de la ventana de juego.

Los parámetros principales del método *Draw()* son los siguientes:

- *Texture2D*: Representa la textura a dibujar. Debe haberse cargado a partir de una imagen en el método LoadContent() de su correspondiente clase.
- *Rectangle*: Posición de la textura a dibujar en la resolución por defecto.
- *Float*: Escalado de la textura que se debe aplicar a la hora de dibujarla. Valores menores que 1 la encogen, los mayores la estiran.
- *Color*: Color a aplicar a la textura a dibujar.

La clase estática ***Font*** permite manejar la salida de texto por pantalla y contiene los diferentes tipos de letra existentes en el juego. Para ello es necesario agregar en la carpeta *Content* del explorador de soluciones archivos con extensión *.spritefont* que contienen el tipo de letra deseado. Posteriormente con el método *LoadContent()* se cargan dichas letras en el atributo de tipo *SpriteFont[]* que contiene esta clase, de forma que ya estaría listo para usarse ese tipo de letra. Hay que destacar que la clase *SpriteFont* es proporcionada por el propio *XNA* para permitir el tratamiento de texto en los juegos.

La funcionalidad principal de esta clase vuelve a ser la de mostrar elementos por pantalla, pero en este caso serán textos. Para ello contiene también métodos *Draw()* que a su vez realizan una llamada al método *DrawString()* de la clase *SpriteResizer*, heredado de la clase *SpriteBatch*. Este método también requiere una serie de parámetros que son los siguientes:

- *SpriteResizer*: Se trata del SpriteBatch usado para dibujar el texto, ya que tal y como está diseñado el código debe recibirse por parámetro.
- *FontStyle*: Enumerado que indica el tipo de fuente a utilizar.
- *Vector2*: Posición de la pantalla donde dibujar el texto.
- *String*: Texto a dibujar por pantalla.
- *float:* Escalado que se debe aplicar al texto a la hora de dibujarlo. Valores menores que 1 encogen la letra, los mayores la agrandan.
- *Color:* Color a aplicar a la fuente.

4.4.3 Módulo de componentes

El módulo de componentes pertenece a la capa de núcleo y permite encapsular los datos de los diferentes objetos que se muestran por pantalla de forma que todos tengan ciertas características y funcionalidades comunes, facilitando el tratamiento y uso de los mismos. Este módulo posee el siguiente diagrama de clases:

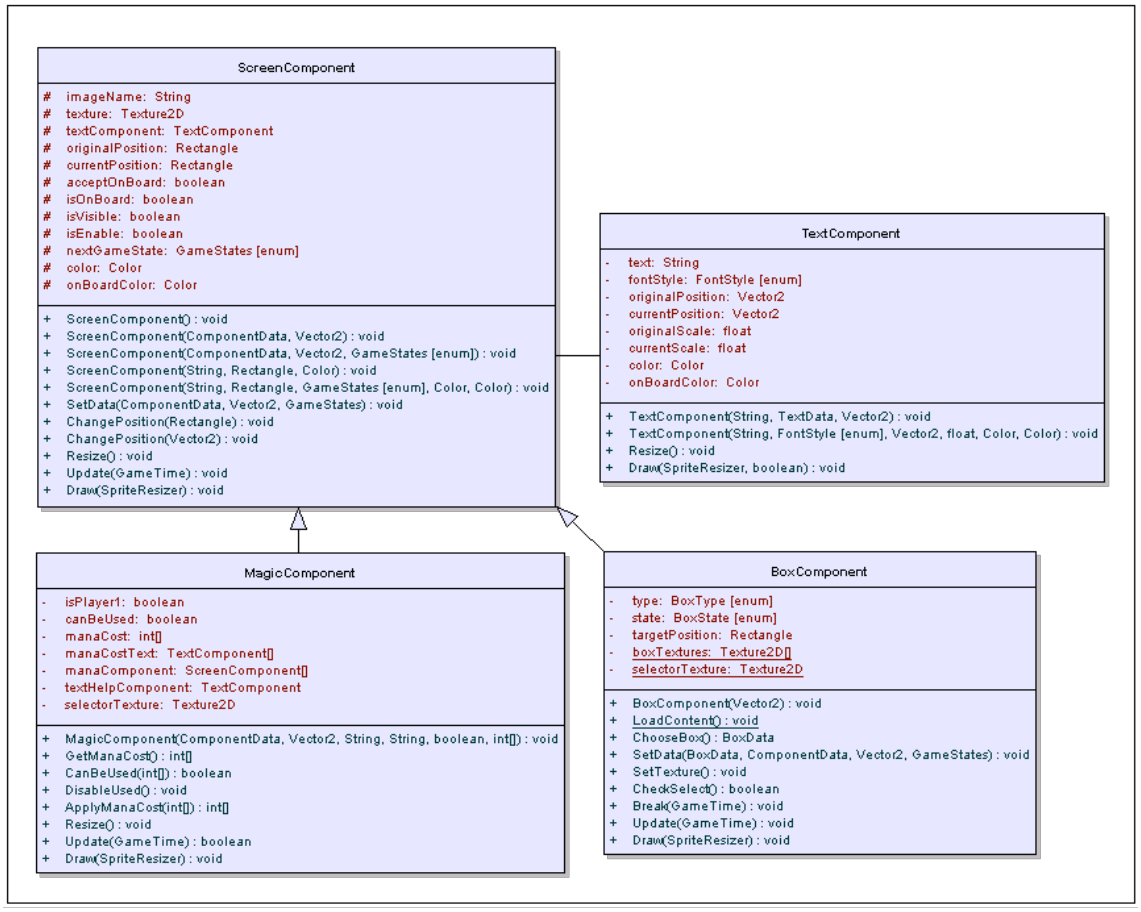

Ilustración 4-6: Módulo de componentes

Se trata de un módulo más complejo que los anteriores, ya que pertenece a la capa núcleo del juego. Está formado por 4 clases que permiten encapsular los datos de diferentes tipos de objetos, y son las siguientes: *ScreenComponent*, *MagicComponent*, *BoxComponent* y *TextComponent*.

Cabe mencionar que las clases de este módulo tienen una gran relación con las del módulo de salida anteriormente descrito, ya que la funcionalidad principal de estas clases es la de encapsular los datos de los diferentes objetos que se muestran por pantalla de forma que todos tengan ciertas características y funcionalidades comunes, y una de dichas funcionalidades es la de dibujar dichos elementos por pantalla, por lo que se realizarán llamadas al método *Draw()* de las clases del módulo de salida para llevar a cabo dicha labor. En los siguientes párrafos se especifica más detalladamente cada una de las clases del módulo.

La clase ***ScreenComponent*** es la más representativa y genérica de todas las clases de este módulo. Su objetivo es encapsular los datos de cualquier tipo de objeto que no tenga ninguna característica en especial, siendo usada generalmente para los botones que aparecen en las distintas pantallas del juego. Guarda información útil para controlar el estado y la apariencia del objeto al que representa. Esta información viene determinada por los siguientes atributos:

- *imageName*: Nombre de la imagen asociada a dicho componente. Para los botones, por ejemplo, la imagen consiste en el marco del mismo.
- *texture*: Textura2D correspondiente a la imagen anterior. Se carga en el método *LoadContent()* de esta misma clase.
- *textComponent*: Componente de texto asociado a este componente. En el caso de que no tenga ningún texto será *null*.
- *originalPosition y currentPosition:* Posiciones del componente dentro de la pantalla de juego para las dimensiones por defecto y con respecto al redimensionamiento de la ventana actual, respectivamente.
- *acceptOnBoard, isOnBoard:* El primero de ellos indica si el componente es susceptible a cambios si tiene situado encima el ratón, mientras que el segundo guarda si en el momento actual el ratón está situado encima.
- *isVisible:* Indica si el componente se debe o no dibujar por pantalla.
- *isEnable:* Indica si el usuario puede interactuar con el componente, es decir, si tiene algún efecto al seleccionarlo con el teclado, pulsar sobre él con el ratón, etc.
- *nextGameState:* Enumerado que indica el siguiente estado al que se pasará del juego en el caso de que se interactúe sobre este componente. Generalmente es usado para cambiar de estado al pulsar algún botón de forma que implique un cambio de pantalla en el juego.
- *color y onBoardColor:* Color a aplicar por defecto al componente a la hora de dibujarlo y color a aplicar en el caso especial de que el ratón se encuentre situado encima de dicho componente.

Como se puede observar, se guarda una gran cantidad de información para cada componente, persiguiendo como objetivo formar las distintas pantallas del juego como un conjunto de dichos componentes, con una apariencia y comportamiento establecidos. De esta forma, únicamente será necesario que para la pantalla de juego en la que se encuentre la partida se recorra la lista de componentes de la misma y se dibujen por pantalla y se realicen sus acciones determinadas de acuerdo a sus datos establecidos. La información necesaria para crear estos componentes se guarda en estructuras del tipo *ComponentData* en el módulo de configuración de forma que, como se mostrará más adelante en dicho apartado, se permita a los desarrolladores modificar las características de los distintos componentes del juego solo con modificar dichos datos.

Las funcionalidades que se pueden desempeñar con estos componentes vienen determinadas por los siguientes métodos:

- *Constructores*: Existe una gran cantidad de constructores diferentes para esta clase, resultando más versátil la forma de introducir información. Una gran cantidad de ellos aceptan estructuras de tipo *ComponentData* mientras que otros no reciben la información en forma de dichas estructuras sino en un conjunto mayor de parámetros.
- *SetData()*: Se trata de un método con una funcionalidad parecida a los constructores, ya que su función es la de introducir diversa información del componente.
- *ChangePosition()*: Permite cambiar los datos relativos a las posiciones del componente dentro de la pantalla de juego; este cambio se realiza tanto para las dimensiones por defecto, como para las dimensiones de la ventana de juego actual. Este método está sobrecargado ya que admite que el parámetro introducido para definir la nueva posición sea tanto de tipo *Rectangle* como de tipo *Vector2*.
- *Resize()*: Lleva a cabo el redimensionamiento del componente, modificando el valor del atributo *currentPosition*, que indica las dimensiones con respecto a la ventana actual. Para ello utiliza el método *Resize()* que ofrece la clase *SpriteResizer*, tal y como se ha explicado anteriormente.
- *Update(Gametime)*: Permite actualizar la lógica del componente para comprobar si se ha realizado alguna acción sobre él y si, por lo tanto, se debe producir algún efecto. Su funcionamiento se basa en comprobar si el componente en cuestión es visible y está habilitado. En caso afirmativo se comprueba si se ha pinchado con el ratón en él y si también ha sido así, se produce el efecto de dicho componente, que se manifiesta cambiando el estado del juego a aquél guardado en el atributo *nextGameState* del componente. Este cambio de estado en muchas ocasiones implica un cambio pantalla en el juego.
- *Draw(SpriteResizer)*: Dibuja el componente por pantalla teniendo en cuenta la información del mismo y siempre y cuando el componente en cuestión sea visible. Adicionalmente en caso de que el ratón se encuentre encima del componente y el componente sea susceptible a este hecho, cambia el color a aplicar a la hora de dibujarlo. En caso de que el *ScreenComponent* también tenga un *TextComponent* asociado, llama al método de dibujar de éste para mostrar también el texto por pantalla. El mecanismo utilizado para dibujar el componente por pantalla es mediante el método *Draw()* de la clase *SpriteResizer* del módulo de salida, introduciendo como parámetros los valores correspondientes a la información del componente.

La clase ***MagicComponent*** hereda de *ScreenComponent* con la finalidad de aportar no solo la información de esta sino información adicional de gran utilidad para los botones de magia de los jugadores. De esta forma aporta nuevas funcionalidades y apariencia a dichos botones, permitiendo establecer nuevos comportamientos en cuanto a su uso y acciones. La nueva información que guarda este tipo de componentes está determinada por los siguientes atributos:

- *isPlayer1*: Indica si se trata de una magia del jugador o del oponente.
- *canBeUsed*: Indica si el jugador posee el maná suficiente como para poder usar la magia. La activación de este atributo no solo implica aspectos funcionales, ya que evidentemente determina si se puede utilizar o no la magia; sino también aspecto visuales, ya que está establecido un cambio en el color del marco y de la letra de las magias que se puede utilizar para proporcionar mayor información al jugador.
- *manaCost[] y manaCostText[]*: Coste de los diferentes manás necesarios para poder utilizar la magia que aparecen debajo del nombre de la misma. El segundo de ellos guarda dicha información en objetos de tipo TextComponent para establecer una apariencia a la hora de mostrar dichos valores por pantalla.
- *manaComponent[]*: Componentes que guardan las imágenes de los diferentes colores de maná asociados a los distintos costes.
- *textHelpComponent*: Texto de ayuda de la magia que se muestra en una caja de texto ubicada debajo del tablero, siempre que el jugador sitúe el ratón encima de la correspondiente magia.
- *selectorTexture*: Textura correspondiente al marco que aparece rodeando a la magia siempre que se pueda usar y sea usada por el jugador.

Se puede ver claramente que la clase *MagicComponent* posee bastante más información que la *ScreenComponent* normal. De hecho, para guardar los componentes correspondientes a las imágenes de los diferentes costes de maná se utiliza de nuevo objetos de tipo *ScreeComponent*, por lo que se trata de un objeto que a su vez contiene objetos de su clase padre. Por otra parte en lo que respecta a las nuevas funcionalidades que poseen este tipo de objetos vienen determinadas por los siguientes métodos:

- *Constructor*: Solo existe un tipo de constructor el cual recibe por parámetro una estructura de tipo *ComponentData*, que será igual para todos los componentes de magias, y una serie de parámetros adicionales que establecen la información adicional necesaria en este tipo de componentes.
- *CanBeUsed(int[])*: Devuelve si se puede usar la magia en función de los valores del status introducido por parámetro. Esta comprobación activará el atributo *canBeUsed* e implicará diferentes características tal y como se muestra en la descripción de dicho atributo.

- *DisableUsed()*: Se deshabilita la posibilidad de poder usar esa magia. Este hecho se puede dar porque se acaba de utilizar y ya no hay suficiente maná o bien porque hay otra magia en juego y temporalmente todas las demás quedan deshabilitadas, o bien por cualquier otra causa pertinente.
- ApplyManaCost(int[]): int[]: Devuelve un nuevo status para el jugador respecto al introducido por parámetro en el cual se decrementan las cantidades de maná correspondientes al uso de esta magia.
- *Resize()*: Sobrescribe el método del padre para llevar también a cabo el redimensionamiento de los componentes adicionales que posee este objeto.
- *Update(Gametime)*: Al igual que el anterior, sobrescribe el método del padre para incorporar una mayor lógica al componente. De esta forma en este caso también se comprueba si se acaba de utilizar la magia, en cuyo caso se hace visible el selector durante un tiempo determinado, se aplica el efecto de la magia correspondiente y se reduce el maná del coste de la misma. También se comprueba si se encuentra el ratón encima de la magia para mostrar el texto de ayuda de la misma.
- *Draw(SpriteResizer)*: Al igual que los dos anteriores, sobrescribe el método padre para dibujar también los objetos de los componentes adicionales que posee. Además posee ciertas características nuevas, como el hecho de establecer un color del marco y de letra diferentes en función de si se puede o no se puede utilizar dicha magia.

La clase ***BoxComponent*** también hereda de *ScreenComponent* tal y como la anterior, pero con la finalidad de aportar información adicional para componentes que van a representar una casilla del tablero. En este caso se guardará información relativa al tipo y estado de la casilla de forma que el tablero pueda interpretar y trabajar con ellas. Los atributos específicos de este tipo de componentes son:

- *type*: Enumerado que indica el tipo concreto de casilla. Los diferentes tipos de casillas que existen para la modalidad principal del juego son: *Damage*, *Maná*, *Red*, *Yellow*, *Green*, *Blue* y *Waste*. El tipo de casilla no solo se evalúa a la hora de detectar si se ha producido una línea, sino que ciertos tipos de casillas producen efectos adicionales a la hora de producir una línea.
- *state*: Enumerado que indica el estado en el que se encuentra la casilla, de forma que se pueda establecer un control sobre la misma. Entre los valores que guarda están, por ejemplo, el estado *Active* que indica que se puede interactuar con ella, el estado *Selected* que indica que la casilla está seleccionada por uno de los jugadores, o bien el estado *Breaking* que indica que la casilla se encuentra en mitad de un proceso de borrado, entre otros estados.

- *boxTextures[]*: Arrays con el conjunto de las texturas de todos los tipos distintos de casillas que existen. Se trata de un atributo estático de forma que permita una única carga de todas las texturas con el método *LoadContent()* para posteriormente usar la correspondiente textura en cada casilla en concreto.
- *selectorTexture*: Textura correspondiente al marco que aparece rodeando a la casilla siempre que sea seleccionada por el jugador.

Los componentes de casillas aportan con respecto a los genéricos de la clase *ScreenComponent* nuevas funcionalidades, de forma que ayuden a su control y manejo por parte del tablero. Dichas funcionalidades vienen determinadas por los siguientes métodos:

- *Constructor*: Permite generar una casilla del tablero. Únicamente es necesario introducir como parámetro la posición de la misma ya que el resto de datos ya están especificados en una estructura de tipo *BoxData* de la clase *Config.* Tampoco es necesario introducir el tipo de casilla a generar ya que esto se realiza al azar con la finalidad de establecer partidas diferentes en cada juego.
- *ChooseBox():BoxData*: Selecciona al azar un tipo de casilla determinada teniendo en cuenta la frecuencia de aparición de la misma. Guarda los datos relativos al tipo de casilla elegido en una estructura de tipo *BoxData* que será usada por el constructor para generar el objeto *BoxComponent*.
- *CheckSelect(): boolean*: Actualiza el estado de la casilla en lo que respecta a la selección de la misma. De este modo en caso de pulsar sobre una casilla que no está seleccionada pasa a estar seleccionada y viceversa.
- *Break():* Establece el estado de la casilla como *Breaking* para indicar que la casilla comienza el proceso de borrado, que será actualizado en el método *Update()* de la misma, tal y como se describirá a continuación.
- *Update():* Este método sobrescribe al establecido por el padre para incorporar una mayor lógica al componente. En concreto además de las funcionalidades del mismo también comprueba el estado de la casilla y en caso de encontrarse como *Breaking* establece un temporizador para producir un efecto de desvanecimiento de la casilla hasta borrarla completamente.
- *Draw():*Al igual que el anterior, sobrescribe el método padre para dibujar adicionalmente el selector de la casilla en el caso de que ésta se encuentre en el estado *Selected*, que indica que se encuentra seleccionada. El color de dicha textura variará según si se trata de una selección por parte del jugador 1 o del jugador 2.

Por último la clase ***TextComponent*** tiene como función encapsular los datos relativos a los textos que se van a mostrar por pantalla. En este caso guarda información como puede ser el tipo letra, tamaño, posición y color, estableciendo además una serie de comportamientos asociados respecto a ciertas acciones por parte del jugador. Esta clase tiene relación con la clase *ScreenComponent* ya que los objetos de esta última pueden tener asociado un objeto de tipo *TextComponent*. De esta forma, por ejemplo, el sistema permite definir un botón de una pantalla como un objeto de tipo *ScreenComponent*, cuyo texto que aparece en el botón esté encapsulado en un objeto de tipo *TextComponent*. Los atributos que definen los objetos de este tipo son los siguientes:

- *text*: Texto asociado a este componente, cuya función es ser mostrado por pantalla.
- *fontStyle*: Enumerado que indica el tipo de fuente a utilizar.
- *originalPosition y currentPosition:* Posiciones del componente dentro de la pantalla de juego para las dimensiones por defecto y con respecto al redimensionamiento de la ventana actual, respectivamente.
- *originalScale y currentScale*: Tamaño de la letra para las dimensiones por defecto y con respecto al redimensionamiento de la ventana actual, respectivamente.
- *color y onBoardColor:* Color a aplicar por defecto al componente a la hora de dibujarlo y color a aplicar en el caso especial de que el ratón se encuentre situado encima de dicho componente.

Los métodos de los que consta este tipo de componentes son:

- *Constructores*: Existen 2 tipos de constructores en los cuales hay que introducir por parámetro el texto del componente para ambos, mientras que se diferencian en que el primero de ellos admite el resto de información en una estructura de tipo *TextData* y el otro recibe dicha información a través de un conjunto de parámetros.
- *Resize()*: Lleva a cabo el redimensionamiento del componente, modificando el valor de los atributos *currentPosition y currenScale*, que indican respectivamente la posición y el tamaño del texto con respecto a la ventana actual. Al igual que para los otros tipos de componentes, se hace uso del método *Resize()* que ofrece la clase *SpriteResizer*.
- *Draw(SpriteResizer)*: Dibuja el texto del componente por pantalla teniendo en cuenta la información del mismo. Adicionalmente en caso de que el ratón se encuentre encima del componente y el componente sea susceptible a este hecho, cambia el color a aplicar a la hora de dibujarlo. El mecanismo utilizado para dibujar el componente por pantalla es mediante el método *Draw()* de la clase *Font* del módulo de salida, introduciendo como parámetros los valores correspondientes a la información del componente.

4.4.4 Módulo de juego

El módulo de juego pertenece a la capa de núcleo y desempeña la labor principal del juego, ya que actúa como controlador del sistema de combate. El diagrama de clases correspondiente a este módulo es el siguiente:

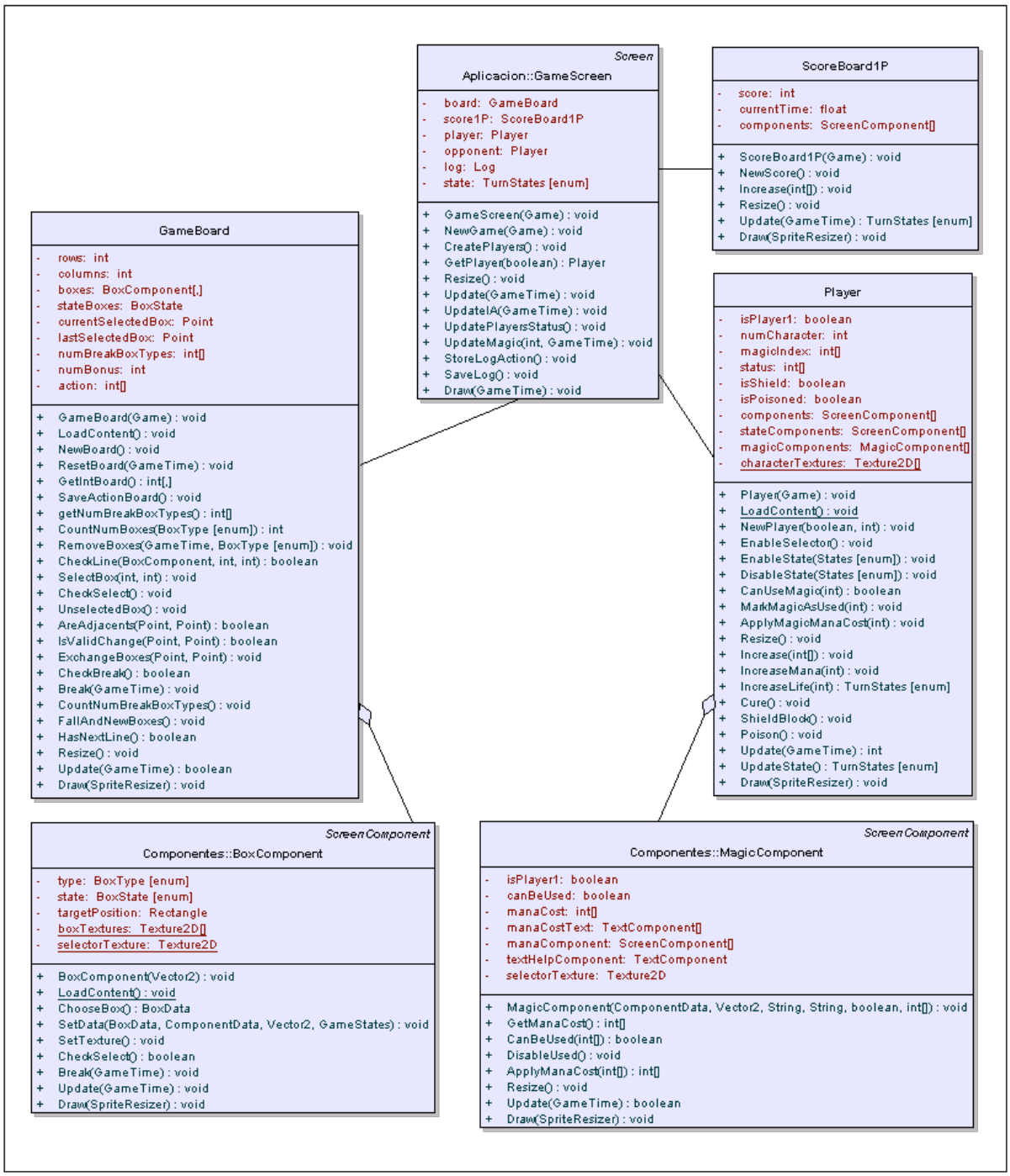

Ilustración 4-7: Módulo de juego

Se puede considerar que este módulo está formado tanto por un conjunto de clases características del mismo, como por clases que pertenecen a otros módulos, pero que también desempeñan una gran labor dentro de este. Así pues, las clases que se corresponden completamente a este módulo son: *GameBoard*, *Player* y *ScoreBoard1P*. No obstante las clases pertenecientes a otros módulos pero imprescindibles para el correcto funcionamiento del sistema de combate son: *GameScreen*, *BoxComponent* y *MagicComponent*.

Hay que destacar que el hecho de que estas últimas clases pertenezcan a otros módulos se debe principalmente a una decisión meramente organizativa, ya que dichas clases podrían pertenecer perfectamente a este módulo pero se ha considerado más conveniente estructurar las clases teniendo en cuenta otros factores, agrupando por una parte las pantallas de juego, como se ha realizado en el módulo de aplicación en el que se encuentra *GameScreen*; y de agrupar un módulo de componentes, como ocurre con *BoxComponent* y *MagicComponent*.

En los siguientes párrafos se expondrán cada una de las clases de las que consta este módulo, describiendo detalladamente todas las clases principales del mismo y comentando las funcionalidades de las otras clases que atañen al funcionamiento de este módulo.

La clase ***GameScreen*** se ha establecido como del módulo aplicación debido a que representa una pantalla del juego, las cuales son gestionadas por el mismo. No obstante, en la práctica esta clase tiene mayor relación con el módulo de juego, ya que actúa como contenedor de todos los elementos pertenecientes al mismo y como organizador principal, gestionando la lógica y el funcionamiento del sistema de combate. Por lo tanto, se procederá a realizar una descripción detallada de la misma en este punto, comenzando por sus atributos:

- *board*: Tablero del juego. Se trata del módulo central del juego, en torno al cual se desarrolla el combate.
- *score1P*: Marcador utilizado para el modo 1 jugador. Consta de un marcador y una barra de tiempo.
- *player y opponent*: Jugadores necesarios en el resto de modos de juego. Cada jugador se caracteriza por poseer un estatus, un estado y una serie de magias.
- *log:* Elemento cuya finalidad es la de generar los diferentes logs que existen para el desarrollo del combate en cada jugada.
- *state*: Enumerado que indica el estado en el que se encuentra la partida, de forma que se pueda establecer un control sobre la misma. Entre los valores que guarda está por ejemplo, el estado *Active* que indica que la partida se está desarrollando con normalidad, o bien el estado *End* que indica que la partida ha finalizado y se debe comenzar una nueva.

Además de los atributos expuestos, característicos de esta clase, posee otro conjunto de atributos al tratarse de un hijo de clase *Screen*. Estos atributos se explicarán posteriormente en dicha clase en el módulo correspondiente, pero cabe destacar uno de ellos, llamado *components[]*, que se trata de un array que contiene los componentes que aparecen en esta pantalla, que en este caso son algunos botones que aparecen en la misma y algunas imágenes especiales que sólo aparecen en determinados momentos de la partida por lo que permanecerán ocultas hasta que llegue el momento, como ocurre con la imagen de *GameOver*.

Los métodos de esta clase permiten llevar a cabo un correcto desarrollo del combate, estableciendo un control sobre los distintos elementos que actúan en el mismo. Se consideran como principales los siguientes métodos:

- *Constructor*: Basándose en la clase *Screen*, de la cual hereda, establece todos los elementos necesarios para crear una pantalla. En este caso, al tratarse de la pantalla de combate, genera el tablero, los jugadores y una serie de componentes característicos de la misma.
- *NewGame(Game)*: Crea una nueva partida dependiendo del tipo de juego seleccionado, estableciendo el estado inicial de todos los elementos del mismo. De esta forma, para el *modo Vs*, genera un nuevo tablero, nuevos jugadores, reinicia el turno, y reestablece el estado de algunos componentes ya que, por ejemplo, se oculta la imagen de *GameOver*.
- *CreatePlayers()*: Crea y asigna los personajes del jugador y del oponente. El juego está diseñado para admitir distintos tipos de jugadores con diferentes características y magias, y este método permite asignar al jugador y al oponente un determinado personaje. El método está preparado para que dicha elección pueda ser aleatoria, aunque actualmente a cada uno de los jugadores se le asigna uno de los dos únicos personajes que actualmente existen.
- *GetPlayer(boolean): Player*: Devuelve el jugador del turno actual en caso de que el parámetro sea true y viceversa. Este método es útil para permitir realizar acciones sobre el jugador del turno indicado por parámetro para alterar su estatus o estado cada vez que se realice un movimiento del tablero o se utilice una magia.
- *Update(GameTime)*: Se trata de uno de los métodos más importantes del juego ya que establece la lógica del combate, realizando una gran cantidad de acciones y comprobaciones. Se encarga de actualizar el tablero y los jugadores, llamando a sus respectivos métodos *Update()* y comprobando si en ese instante se ha realizado alguna línea o una magia. En caso positivo se llama a los métodos *UpdatePlayersStatus()* ó *UpdateMagic(int, GameTime)*, cuyo comportamiento se explicará a continuación, y cuyo objetivo es el de actualizar el estado de los jugadores en relación al movimiento del tablero o a la magia utilizada, respectivamente.

Este método también se encarga de controlar al jugador automático en caso de que exista, estableciendo si se trata de un turno en el que tiene que llevar a cabo alguna acción, llamando en caso de que así sea al método UpdateIA(GameTime), que se explicará también a continuación de este método, y que permitirá obtener una acción por parte de la IA.

Otra funcionalidad que lleva a cabo este método es la del control en la generación del log, ya que envía al módulo encargado del mismo la información necesaria cada vez que se produce alguna acción en el combate, de forma que quede reflejado completamente todas las acciones y estados que tienen lugar en el combate. Establece el momento en que se debe guardar el log, llamando a los métodos *StoreLogAction()* y *SaveLog()*.

En el caso del modo *1 jugador* establece un comportamiento especial, ya que no tiene en cuenta todo lo mencionado anteriormente a cerca de los jugadores y, sin embargo, actualiza y comprueba el estado del marcador en cada instante, a través de su método *Update()*. De esta forma incrementa el marcador y el tiempo restante cada vez que el usuario genera una línea, y reduce el tiempo restante en caso de que no sea así, finalizando la partida en caso de que dicho tiempo llegue a 0.

Por último este método establece el control de la finalización del combate, comprobando para cada acción si la vida de alguno de los jugadores se reduce a 0. En caso de que así sea bloquea el tablero y los jugadores para que el usuario no pueda realizar ninguna acción sobre los mismos, y elimina todas las casillas del tablero, mostrando acto seguido una imagen que indica el jugador vencedor. La única acción posible en ese momento es la de utilizar el botón de salir para volver a la pantalla principal y desde ahí comenzar un nuevo combate.

- *UpdateIA(GameTime)*: Se encarga de actualizar el juego para el *modo Vs* en el turno de un jugador controlado por la IA. Para ello prepara todos los datos del estado actual del combate y llama con ellos al método *Run()* del jugador automático, que será el encargado de devolver una acción determinada. Dicha acción será interpretada por este método, encargándose de trasladarla al módulo correspondiente y llevar a cabo el movimiento del tablero o realizar el uso de la magia oportuna. Además, debido a que el módulo del jugador automático se considera externo al sistema del juego y está preparado para ser modificado por los usuarios, este método debe llevar un control sobre el mismo, verificando que la acción realizada es una acción correcta y permitida. En caso de que no sea así se devuelve de nuevo el control a dicho módulo para esperar una nueva acción y si finalmente después de varias veces no devuelve una acción permitida, se considera que ha perdido su turno. Por último este módulo también lleva a cabo un control sobre el tiempo de respuesta de la IA, realizando los movimientos con cierto intervalo de tiempo para permitir a los jugadores humanos detectar la jugada correctamente.

- *UpdatePlayerStatus()*: Actualiza el estatus de los jugadores después de cada movimiento de tablero, aumentando el maná correspondiente del jugador del turno actual respecto a las líneas que haya realizado, y disminuyendo la vida del jugador contrario en caso de que alguna de las líneas producidas sea de casillas de tipo *Damage*.
- *UpdateMagic(int, GameTime)*: Actualiza el estatus de los jugadores después de utilizar una magia, comprobando el tipo de magia realizada, cuyo índice recibe por parámetro, y en función de ello se encarga de llamar al correspondiente método de dicha magia o de modificar directamente los valores de los estatus de los jugadores.
- *StoreLogAction() y SaveLog()*: Permiten al módulo del log llevar a cabo su funcionalidad, enviándole la información necesaria cada vez que se produce alguna acción en el combate, de forma que queden reflejadas completamente todas las acciones y estados que tienen lugar en el combate.
- *Draw(GameTime)*: Dibuja todos los elementos relacionados con la pantalla de combate. En este caso no necesita recibir por parámetro el *SpriteResizer* ya que esta clase actúa como raíz y es la que contiene dicho manejador gráfico.

La clase ***GameBoard*** representa el tablero de juego utilizado para el combate, encargándose de la gestión de su lógica y correcto funcionamiento. El tablero de juego se utiliza tanto en el modo *1 jugador* como en el *modo Vs*, adquiriendo diferentes matices en cada caso como, por ejemplo, el tipo de casillas que aparecen en cada modo. El tablero queda definido por los siguientes atributos:

- *rows y columns*: Indica las filas y las columnas, en número de casillas, que tiene el tablero.
- *boxes[,]*: Array bidimensional que guarda todos los componentes de casilla del tablero. Posee una particularidad, y es que para ajustar la representación por pantalla de cada una de las casillas con las coordenadas, se han invertido los ejes de estas, de forma que, por ejemplo, la casilla [1,3] pasa a ser la [3,1] y viceversa. De esta forma se simplifica el funcionamiento de algunos métodos y la representación por pantalla es más directa con respecto a los datos guardados.
- *stateBoxes*: Enumerado que indica el estado en el que se encuentran en general las casillas. Puede tomar los mismos valores que el estado que tomaba cada casilla en concreto, siendo el valor de este el más restrictivo de todos, es decir, si todas las casillas se encuentran en estado *Active* y solo una de ellas en estado *Breaking*, este atributo tomará el valor de ese último estado.
- *currentSelectedBox y lastSelectedBox*: Guardan respectivamente las posiciones de la casilla seleccionada actualmente y anteriormente. Son de utilidad a la hora de realizar los movimientos de las casillas.

- *numBreakBoxTypes[]*: Array que guarda el número de casillas de cada tipo que se han eliminado en la última jugada. Este array será consultado posteriormente por el método *Update()* de la clase *GameScreen*, anteriormente descrita, para incrementar los valores del estatus del jugador correspondiente.
- *numBonus*: Valor de bonus de maná adicional que se ha producido gracias a la última jugada. Se obtiene un bonus de 1 y 3 cada vez que se producen líneas de 4 y 5 casillas respectivamente, para premiar al jugador por realizar este tipo de movimientos. El valor de este atributo tiene la misma funcionalidad que el array anterior.
- *action[]:* Array en el que se guarda la acción que realiza el jugador del turno actual en el juego, y que será enviada posteriormente al módulo del log para que quede registrado dicho movimiento.

Esta clase posee una gran cantidad de métodos de forma que dote de un correcto funcionamiento el tablero de juego. Los más importantes son:

- *Constructor*: Genera un nuevo tablero de juego en función de las dimensiones del mismo.
- *NewBoard()*: Prepara el tablero para una nueva partida, generando de nuevo aleatoriamente todas las casillas, llevando a cabo un control para evitar que este hecho produzca que se forme una línea inicialmente. Además reestablece otros valores como el atributo *stateBoxes* y las casillas seleccionadas.
- *ResetBoard(GameTime)*: Reinicia el tablero borrando todas las casillas y generándolas de nuevo.
- *GetIntBoard(): int[]*: Devuelve una versión simplificada del tablero, generando una representación numérica del mismo para cada tipo de casilla. Los datos obtenidos serán de utilidad para utilizarlos en el módulo de *log* y en el de *IA*.
- *SaveActionBoard()*: Guarda en el atributo *action[]* la acción que realiza el jugador del turno actual en el juego, y que será enviada posteriormente al módulo del log para que quede registrado dicho movimiento.
- *CountNumBoxes(BoxType): int*: Devuelve el número de casillas que tiene el tablero del tipo indicado por parámetro.
- *RemoveBoxes(GameTime, BoxType)*: Elimina todas las casillas del tablero del tipo indicado por parámetro.
- *CheckLine(BoxComponent, int, int): boolean*: Comprueba si la casilla introducida por parámetro en la posición determinada por los otros 2 parámetros, generaría una línea actualmente en el tablero. Este método es consultado a la hora de crear un nuevo tablero para evitar que se genere directamente con líneas.

- *CheckSelect():* comprueba el estado de seleccionadas de cada una de las casillas mediante una llamada al método *CheckSelect()* de la própia casilla. Actualiza los valores de *currentSelectedBox y lastSelectedBox* en caso de que se haya seleccionado alguna casilla.
- *AreAdjacents(Point, Point): boolean:* Comprueba si los dos puntos introducidos son adyacentes. Esta comprobación es necesaria a la hora de intercambiar dos casillas, ya que es una condición que deben de cumplir.
- *IsValidChange(Point, Point) boolean:* Comprueba si se produce alguna línea al intercambiar las dos casillas de los puntos introducidos por parámetro. Es la otra comprobación necesaria a la hora de intercambiar dos casillas, ya que para que un movimiento sea válido, alguna de las casillas a mover debe genera una línea.
- *ExchangeBoxes (Point, Point):* Intercambia las dos casillas de los puntos introducidos por parámetro. Este método es llamado tras comprobar que cumple las comprobaciones anteriores y se trata de un movimiento válido.
- *CheckBreak(): boolean:* Comprueba si existe alguna línea en el tablero y, en caso afirmativo, marca dichas casillas en el estado que señala que deben ser borradas. Este método se utiliza después que el anterior, al intercambiar dos casillas después de haber realizado las comprobaciones pertinentes, hecho que garantiza que exista alguna línea.
- *CountNumBreakBoxTypes():* Guarda en el array *numBreakBoxTypes[]* el número de casillas de cada tipo que se han eliminado en la última jugada.
- *FallAndNewBoxes():* Desplaza las casillas hacia abajo del tablero quitando los huecos que han dejado las casillas que se han borrado anteriormente al formar una línea. De esta forma todos los huecos quedan en la parte superior del tablero, y en ellos se introduce una nueva casilla.
- *HasNextLine(): boolean:* Comprueba si hay en el tablero algún posible movimiento para realizar en la siguiente acción de forma que genere una línea. Este método es útil para controlar periódicamente que el tablero no se quede sin movimientos y, si fuera oportuno, volver a generarlo con el método explicado anteriormente *ResetBoard(GameTime)*.
- *Update(Gametime): boolean*: Actualiza la lógica del tablero, ayudándose para ello del atributo *stateBoxes* que indica el estado en el que se encuentran en general las casillas, y de gran parte de los métodos anteriormente descritos. Se trata de un método que comprueba continuamente si algún jugador ha realizado algún movimiento y, en ese caso, comprueba que sea correcto y que produzca línea. Si es así, se borran las casillas que produzcan línea y se eliminan los huecos

desplazando las casillas superiores hacia abajo e introduciendo nuevas casillas. Por último se comprueba si se ha producido una línea en cascada, es decir, si al bajar las casillas para eliminar huecos se han producido nuevas líneas. En ese caso se procede a eliminar también dichas casillas y se repite el proceso tantas veces como sea oportuno hasta que no se produzca ninguna nueva línea. Finalizado el proceso, se devuelve el número de casillas obtenidas de cada tipo, para actualizar el estatus del jugador, y se cambia de turno. Adicionalmente se realiza una última comprobación que consiste en determinar si existe algún movimiento para realizar en la siguiente acción de forma que genere una línea y, si no es así, se resetea el tablero.

La clase ***BoxComponent*** está estrechamente relacionada con la clase *GameBoard* que se acaba de describir, hasta tal punto que esta segunda está formada por un agregado de objetos de la primera. *BoxComponent* es a su vez una clase hijo de *ScreenComponent* y pertenece al módulo *componentes*, por lo que se puede consultar sus características en la descripción realizada en dicho módulo.

La clase ***Player*** representa cada uno de los jugadores que se enfrentan en el combate. Es utilizada tanto para jugadores humanos como para jugadores controlados automáticamente, ya que su finalidad principal es la de llevar a cabo un control de su estatus y magias, reconociendo las acciones realizadas para ambos tipos de jugadores. Los atributos que posee esta clase son los siguientes:

- *isPlayer1*: Indica si se trata del jugador o del oponente.
- *numCharacter*: Índice del tipo de personaje que le corresponde a este jugador.
- *magicIndex[]*: Índice del conjunto de magias que posee el personaje. Identifica de entre todas las magias que existen en el juego aquellas que un personaje en concreto posee, haciendo referencia a las mismas.
- *status[]:* Estatus del jugador. Los atributos del estatus que posee cada personaje son: vida, maná, rojo, amarillo, verde y azul.
- *isShield e isPoisoned:* Indican el estado del jugador en lo que respecta a si tiene activado el escudo y si está envenenado.
- *components[]:* Array que contiene los componentes característicos de los que está formado un personaje, como por ejemplo su imagen.
- *stateComponents[]:* Array que contiene los componentes de las imágenes de los estados que pueden afectar a un jugador.
- *magicComponents[]:* Conjunto de los componentes de magia que posee el personaje.
- *characterTextures[]*: Array que guarda las distintas texturas que se corresponden a cada una de las imágenes de los posibles personajes. Existen más atributos en esta clase que guardan también otras texturas.

Los métodos de esta clase aportan la funcionalidad deseada a los jugadores, controlando su estatus y sus magias. Los más importantes son:

- *Constructor*: Genera un nuevo jugador, inicializando los distintos componentes de los que está formado.
- *LoadContent()*: Método estático que carga todas las texturas de todos los posibles personajes solo una vez para que posteriormente cada vez que se cree un personaje se le asigne una de las texturas que ya han sido cargadas.
- *NewPlayer(boolean, int):* Crea un nuevo jugador asignándole un personaje y sus magias correspondientes. Se indica por parámetro si se trata del jugador o del oponente, así como el índice del tipo de personaje que le corresponde a ese jugador. Dependiendo del valor de este parámetro se establece un determinado estatus al jugador y se le asigna un determinado conjunto de magias.
- *EnableSelector():* Activa el marco que rodea la imagen del jugador de forma que se permita al usuario distinguir visualmente cual es el usuario que posee el turno en cada momento.
- *EnableState(States) y DisableState(States):* Permiten activar y desactivar en el jugador el estado que recibe por parámetro. Los estados en los que se puede encontrar en el desarrollo actual un jugador vienen representados por las variables *isShield e isPoisoned*, e indican si el jugador tiene activado el escudo y si está envenenado, respectivamente.
- *CanUseMagic(int): boolean:* Devuelve si el jugador tiene maná suficiente para utilizar la magia cuyo indice es pasado por parámetro.
- *ApplyMagicManaCost(int):* Actualiza el estatus del jugador en el cual se decrementan las cantidades de maná correspondientes al uso de la magia del índice introducido por parámetro.
- *Increase(int[])*: Incrementan el estatus del jugador de acuerdo a los valores recibidos por parámetro. Cantidades negativas indican un decremento en el valor correspondiente del estatus del jugador controlando que nunca pueda ser menor que 0.
- *IncreaseLife(int): Turnstates*: Método que funciona del mismo modo que el anterior pero aplicado únicamente a la vida del jugador. En este caso devuelve un valor de tipo *TurnStates* cuya finalidad es indicar que se ha llegado al estado de *GameOver* si se ha decrementado la vida del jugador y esta ha llegado a 0.
- *Cure(), ShieldBlock() y Poison()*: Magias que puede utilizar el jugador. Generalmente aumentan o reducen su estatus del jugador, o bien activan o desactivan alguno de los posibles estados en los que se puede encontrar, representados por las variables *isShield e isPoisoned.*

- *Update(Gametime): int:* Actualiza la lógica del jugador, actualizando todos los componentes que contiene y comprobando si se ha utilizado alguna magia. En ese caso este método devolverá el valor del índice de la magia utilizada, y -1 en caso contrario.
- *UpdateState(): TurnStates:* Actualiza el estatus del jugador en función de su estado actual. Este método es llamado por la clase *GameScreen* cada vez que es el turno del jugador y permite llevar a cabo acciones como decrementar la vida del jugador en caso de encontrarse envenenado. Este método devuelve un valor de tipo *TurnStates* indicando el estado *GameOver* en caso de que la vida del jugador haya llegado a 0.

La clase ***MagicComponent*** representa cada una de las magias que posee un jugador, estableciendo la clase *Player* un agregado de objetos *MagicComponent*. Se trata de una relación similar a la descrita anteriormente entre las clases *BoxComponent* y *GameBoard*, ya que la clase *MagicComponent* también es a su vez una clase hijo de *ScreenComponent* y pertenece al módulo *componentes*, pudiendo consultarse sus características en la descripción realizada en dicho módulo.

La clase ***ScoreBoard1P*** contiene el estado del marcador y del contador del tiempo, utilizados para el modo *1 jugador*. Se trata de una clase bastante sencilla que posee los siguientes atributos:

- *score*: Guarda la puntuación de la partida actual.
- *currentTime*: Tiempo que le queda al jugador antes de que se termine la partida. Se representa por pantalla mediante una barra de tiempo.
- *components:* Conjunto de componentes que definen el marcador y la barra de tiempo.

Los métodos que contiene esta clase son los siguientes:

- *Constructor*: Crea el marcador, la barra de tiempo, y todos sus componentes relacionados.
- *NewScore()*: Realiza la puesta a 0 de la puntuación del marcador y reestablece la barra de tiempo poniéndola a su valor máximo.
- *Increase(int[])*: Incrementa la puntuación y la barra de tiempo de acuerdo a las casillas eliminadas recibidas por parámetro.
- *Update(): TurnStates*: Actualiza la lógica de la barra de tiempo, decrementándola una cierta cantidad establecida y comprobando si se ha terminado el tiempo, en cuyo caso devuelve un valor de tipo *TurnStates* indicando el estado *GameOver*.
- *Draw()*: Dibuja por pantalla el marcador, la barra de tiempo, y todos sus componentes relacionados.

4.4.5 Módulo de aplicación

El módulo de aplicación pertenece a la capa del núcleo y es el encargado de llevar a cabo la ejecución del juego y controlar los elementos del propio sistema. Establece el control de las distintas pantallas que componen el juego y la transición entre las mismas de acuerdo a las distintas circunstancias y acciones que se realicen en la partida. El diagrama de clases de dicho módulo es el mostrado en la siguiente ilustración:

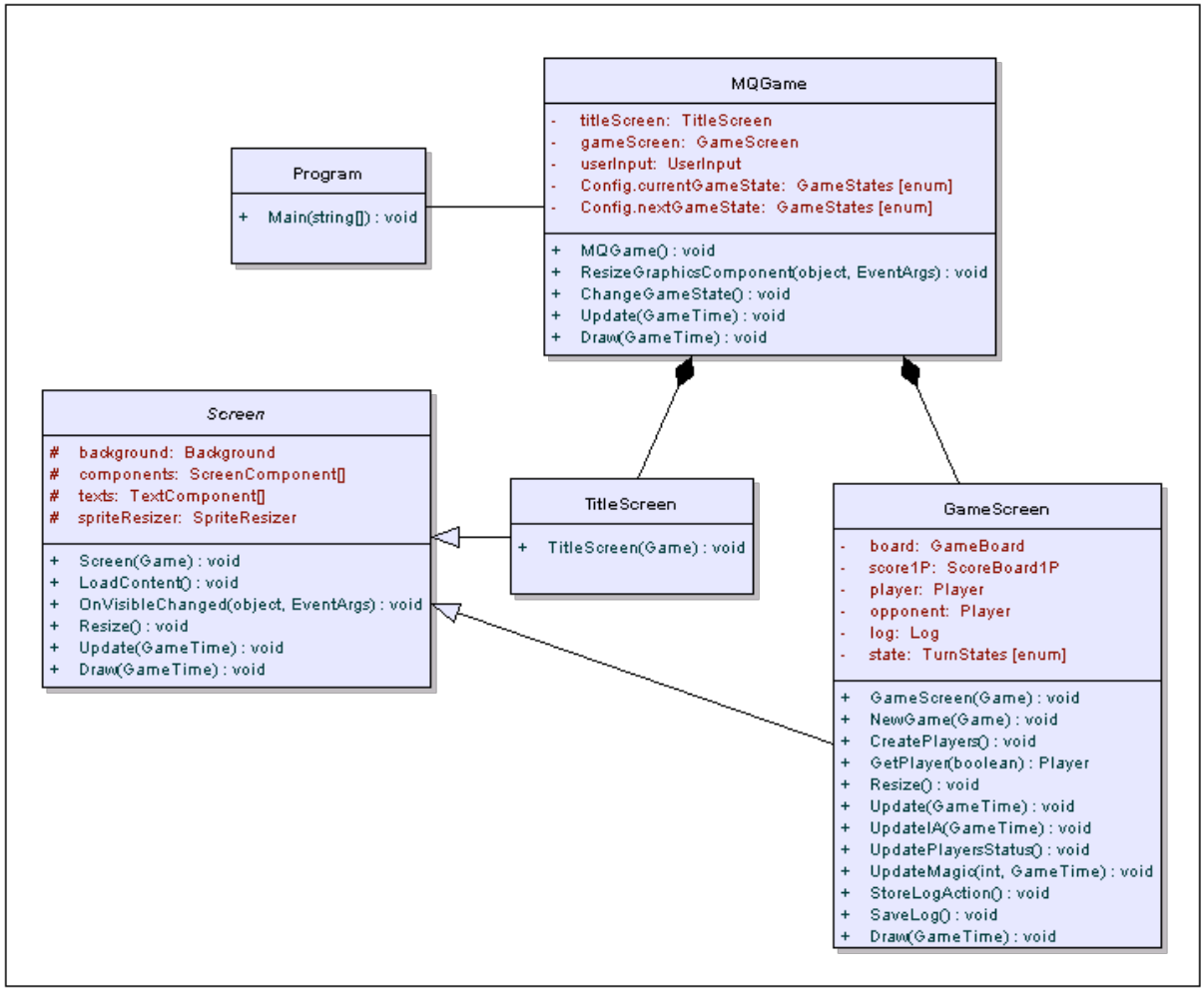

Ilustración 4-8: Módulo de aplicación

Se trata de un módulo que establece las bases y la estructura de la aplicación de una forma sencilla pero potente, controlando los estados del juego y las distintas pantallas del mismo. Como se observa en el diagrama, actualmente consta de 5 clases: *Program*, *MQGame*, *Screen*, *TitleScreen* y *GameScreen*. La primera de ellas, *Program*, simplemente se encarga de lanzar la ejecución del programa llamando a la segunda de ellas, *MQGame*, que controla los estados del juego y las diferentes pantallas de las que consta. *Screen* es una clase abstracta que establece una estructura general que deben cumplir las diferentes pantallas, aportando características y funcionalidades. Las 2 últimas clases son pantallas concretas correspondientes a la pantalla inicial y a la de combate.

La clase ***Program*** únicamente contiene el método *Main()* y su única finalidad es la de lanzar la ejecución del programa, creando un objeto de tipo *MQGame* y llamando al método *Run()* del mismo.

La clase ***MQGame*** lleva un control de las distintas pantallas que componen el juego, estableciendo la transición entre las mismas de acuerdo a las distintas circunstancias y acciones que se realicen en la partida. Para realizar satisfactoriamente dicha función, este módulo guarda una serie de estados que permiten controlar la situación del juego con cada momento. Esta clase viene definida por los siguientes atributos:

- *titleScreen y gameScreen*: Conjunto de pantallas que forman el juego.
- *userInput*: Permite manejar la entrada de datos en el juego, centralizada en un objeto de este tipo, y que permite obtener los datos del ratón.
- *Config.currentGameState y Config.nextGameState*: Estado actual y próximo estado en los que se encuentra el juego. Las modificaciones en los valores de *Config.nextGameState* las producen los objetos de tipo *ScreenComponents* en el momento que son usados. Un cambio en dicho valor implica generalmente una transición entre distintas pantallas.

Esta clase contiene una serie de métodos de control de los distintos elementos del sistema, que se detallan a continuación:

- *Constructor*: Genera un nuevo juego, cargando y configurando el dispositivo gráfico, estableciendo las configuraciones de la ventana inicial, inicializando la entrada de datos y creando las diferentes pantallas de juego para permitir su uso cuando se considere necesario.
- *ResizeGraphicsComponent(objetc, EventArgs)*: Método que captura el evento que indica que se ha redimensionado la pantalla de juego y actúa como origen para la redimensión de todos los elementos del juego. Es el encargado de llamar al método *Resize()* del *SpriteResizer* y de las diferentes pantallas, y éstas a su vez serán las encargadas de llamar a dicho método para sus correspondientes elementos.
- *ChangeGameState()*: Comprueba los valores de los atributos descritos anteriormente *Config.currentGameState y Config.nextGameState* para detectar si se ha producido algún cambio en el estado del juego. En caso afirmativo realiza las acciones pertinentes a dicho cambio de estado, que suele implicar una transición entre distintas pantallas.
- *Update()*: Actualiza la lógica del juego, encargándose además de controlar la pulsación del botón escape para alternar entre pantalla completa y ventana la ejecución del juego, y llamando al método *ChangeGameState*() para controlar el estado del mismo.
- *Draw()*: Este método no dibuja ningún componente en este caso, sino que se encarga de establecer como negro el color del fondo del juego sobre el que se van a dibujar todos los demás objetos.

La clase ***Screen*** es una clase abstracta cuyo objetivo es establecer una estructura general que deben cumplir las diferentes pantallas, aportando características y funcionalidades. Los atributos de los que consta son los siguientes:

- *background:* Toda pantalla debe tener asociada una estructura de fondo que permitirá mostrar dicha imagen en un plano más lejano que todo el resto de componentes que tenga la pantalla.
- *components:* Conjunto de componentes generales que posee la pantalla, tales como los botones de la misma o ciertas imágenes.
- *text:* Conjunto de textos que puede tener la pantalla, como pueden ser los títulos o ciertos menús.
- *spriteResizer:* Manejador gráfico que permitirá dibujar los textos, componentes y el fondo de la pantalla.

Los métodos generales que poseen todos los objetos de este tipo son:

- *Constructor*: Crea una pantalla genérica.
- *LoadContent()*: Crea el *SpriteResizer* que será utilizado para dibujar los distintos elementos de la pantalla y carga el contenido gráfico del fondo de pantalla y de los distintos componentes que contenga.
- *OnVisibleChanged(objetc, EventArgs)*: Método que captura el evento que indica que esta determinada pantalla se ha hecho visible, permitiendo mostrar sus elementos correctamente y, en el caso de que se considere oportuno, realizar acciones adicionales.
- *Resize()*: Redimensiona todos los elementos que contiene, es decir, los textos, componentes y el fondo de pantalla.
- *Update()*: Actualiza la lógica de los componentes que posee la pantalla.
- *Draw()*: Inicia el *SpriteResizer* con el método *Begin()* para dibujar todos los elementos de la pantalla. Terminado el proceso finaliza con el método *End()* de *SpriteResizer*.

La clase ***TitleScreen*** es una representación concreta de *Screen*, correspondiente a la pantalla inicial del juego. Únicamente está formada por un conjunto de componentes específicos, y no contiene ninguna funcionalidad ni elemento de especial relevancia.

La clase ***GameScreen*** es otra representación concreta de *Screen*, correspondiente a la pantalla de combate del juego. Contiene todos los elementos necesarios para llevar a cabo dicha función, estableciendo el tablero y los jugadores, así como el resto de componentes que lo conforman. Esta clase, aunque pertenece al módulo actual, ya ha sido explicada en detalle en el módulo de juego, ya que actúa como contenedor de todos los elementos pertenecientes a dicho módulo, gestionando la lógica y el funcionamiento del sistema de combate.

4.4.6 Módulo de configuración

El módulo de configuración pertenece a la capa de suplementos y permite centralizar los datos de configuración del juego y de los distintos elementos que lo componen. El diagrama de clases de dicho módulo es el mostrado en la siguiente ilustración:

Ilustración 4-9: Módulo de configuración

Este módulo está formado únicamente por la clase estática ***Config***, que contiene los datos de todas las características del juego y está estructurada en un conjunto de regiones que las agrupa por funcionalidad. Esta clase tiene como finalidad permitir la modificación de las características del juego de una forma rápida y sencilla, así como dotar de la posibilidad de introducir nuevos personajes y magias. Las regiones en las que está estructurada esta clase son las siguientes:

- *Resolución de pantalla:* Establece las características de las dimensiones de la ventana de juego.
- *Estructuras Data:* Conjunto de diferentes tipos de estructuras con los datos necesarios para llamar a los constructores de los distintos elementos del juego. Cada estructura contiene una gran cantidad de información que permite asignar los valores a los atributos de la correspondiente clase.
- *Estructuras TextData:* Conjunto de diferentes tipos de estructuras con los datos necesarios para crear los distintos elementos *TextComponent* que aparecen en el juego.
- *Posiciones:* Coordenadas y dimensiones de los diferentes elementos que aparecen en pantalla. A la hora de crear dichos elementos, estos consultarán los valores guardados en esta clase para determinar las posiciones en las que dibujarlos.
- *Nombres de imágenes:* Nombres de la ruta y de las imágenes cargadas dentro del contenedor de soluciones, que se utilizarán en cada uno de los correspondientes métodos *LoadContent()* de las diferentes clases.
- *Colores:* Colores a aplicar a los diferentes elementos que se dibujan por pantalla. Para algunos elementos no hace falta definir el color porque ya está definido dentro de su estructura *Data* correspondiente.
- *Características del juego generales:* Guarda los valores de las características que afectan al tablero y al *modo 1 jugador*. Entre otras se encuentra el tiempo de borrado de las casillas y la velocidad de decremento de la barra de tiempo.
- *Características del juego Vs:* Guarda valores de características que afectan al *modo Vs*. Entre otras guarda las características de los diferentes personajes, los índices sus magias y sus costes asociados.
- *Estados del juego y turnos:* Establece los estados del juego. utilizados por el módulo de aplicación para controlar las transiciones entre las distintas pantallas de juego. También se guarda qué jugador posee el turno y se aporta un método que realiza el cambio de turno.
- *Log:* Conjunto de valores que establecen los diferentes datos necesarios para guardar el log. Establece, entre otros, que tipos de logs se deben guardar y las rutas y nombres de los archivos donde se guardan.

4.4.7 Módulo de log

El módulo de log pertenece a la capa de suplementos y está encargado de generar un registro del desarrollo de los combates a partir de la información recibida por parte del módulo de juego. El diagrama de clases correspondiente a este módulo es el mostrado en la siguiente ilustración:

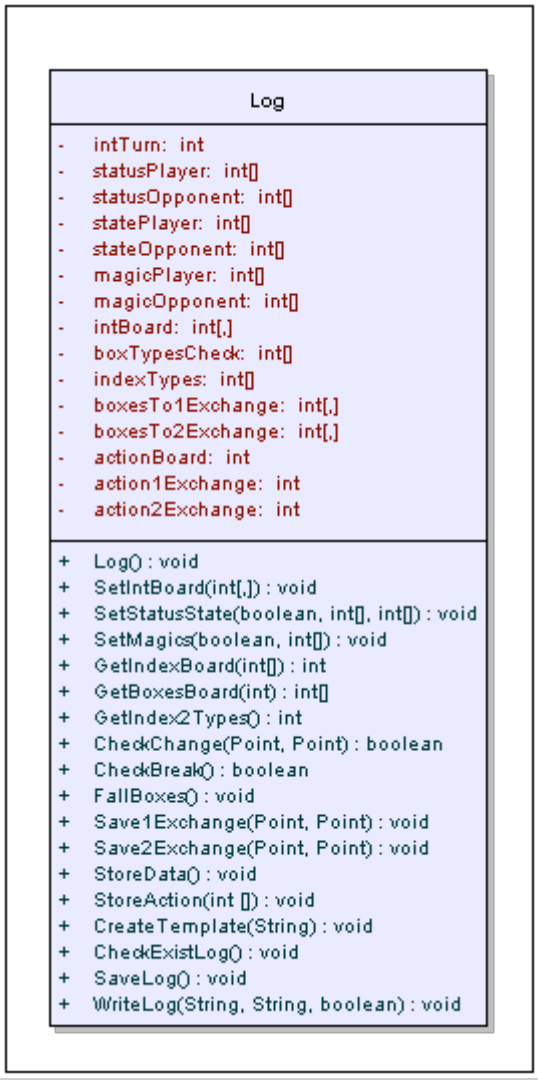

Ilustración 4-10: Módulo de log

Se puede observar que este módulo también consta de una única clase, llamada ***Log***, que guarda una representación simplificada del estado del combate a partir de los datos proporcionados por el módulo de juego en cada acción realizada. Esta clase realiza un tratamiento de dichos datos produciendo nuevos datos derivados a partir de los mismos; con todo ello finalmente se generan 3 tipos de logs diferentes que guardan, con distintos matices, el estado del combate y la acción llevada a cabo en cada turno. Estos datos serán guardados en la ruta "*MagicQuest\bin\x86\Debug\Content\logs*" en archivos con extensión .arff con los siguientes nombres: *logBoard*, *log1Exchange* y *log2Exchange.*

Con la finalidad de guardar una representación simplificada del estado del combate, esta clase contiene los siguientes atributos:

- *intTurn*: Indica con un valor numérico a que jugador pertenece el turno.
- *statusPlayer[] y statusOpponent[]*: Estatus del jugador del turno actual y del oponente a este, respectivamente. Los atributos que se guardan en dicho estatus son: vida, maná, rojo, amarillo, verde y azul.
- *statePlayer[] y stateOpponent[]:* Estados en los que se encuentra el jugador del turno actual y el oponente de este, respectivamente. Para el índice 0 guarda si el jugador tiene el bloqueo de escudo activado, y para el índice 1 guarda si el jugador está envenenado. Se indica con un 1 en caso de que esté activo dicho estado y con un 0 en caso contrario.
- *magicPlayer[] y magicOpponent[]:* Arrays que indican mediante un 1 que el jugador o el oponente respectivamente, tiene suficiente maná como para utilizar dicha magia, y con un 0 en caso de que no pudiera.
- *intBoard[,]*: Representación numérica del tablero. La correspondencia es la siguiente: ataque → 0, maná → 1, rojo → 2, amarillo → 3, verde → 4, azul → 5 y basura → 6.
- *boxTypesCheck[] e indexTypes[]:* Arrays que se utilizan de modo auxiliar para la generación de datos derivados a partir de los valores anteriores.
- *boxesTo1Exchange[,] y boxesTo2Exchange[,]:* Arrays que guardan si se puede obtener un determinado tipo/combinación de tipos, y en caso positivo cuales son las casillas de la fila más baja con las que realizarlo.
- *actionBoard:* Valor que representa numéricamente la acción realizada en un determinado turno codificada para el archivo de log *logBoard*. Tomará valores en el rango [0…4] para acciones correspondientes a las distintas magias. Valores superiores indican un movimiento entre dos casillas adyacentes del tablero. De esta forma cada magia o posible movimiento tiene correspondencia unívoca con el valor guardado en este atributo.
- *action1Exchange*: Valor que representa numéricamente la acción realizada un determinado turno codificada para el archivo de de log *log1Exchange*. Tomará valores en el rango [0…4] para acciones correspondientes a las distintas magias. Valores superiores indican que el jugador ha realizado una línea de un determinado tipo. Los valores concretos de este atributo se comentarán detalladamente en la definición del log *log1Exchange* del apartado 4.5.1 Definición de los logs.
- *action2Exchange*: Valor que representa numéricamente la acción realizada un determinado turno codificada para el archivo de de log *log2Exchange*. Tomará valores en el rango [0…4] para acciones correspondientes a las distintas magias. Valores superiores indican el tipo/combinación de tipos de la línea/líneas que se han realizado. Los valores concretos de este atributo se comentarán detalladamente en la definición del log *log2Exchange* del apartado 4.5.1 Definición de los logs.

Los métodos de esta clase están orientados fundamentalmente a la preparación de los datos que contienen los atributos para generar los distintos tipos de logs existentes. En ese sentido, los métodos más destacados son los siguientes:

- *Constructor*: Genera un nuevo gestor del log, inicializando los arrays que guardan una versión simplificada del estado del combate en cada turno.
- *SetIntBoard(int[,])*: Guarda la representación numérica del tablero. La correspondencia es la siguiente: ataque → 0, maná → 1, rojo → 2, amarillo → 3, verde → 4, azul → 5 y basura → 6.
- *SetStatusState(boolean, int[], int[])*: Guarda para el jugador o para el oponente, dependiendo del booleano recibido por parámetro, los valores de estatus y de estado recibidos en los otros dos parámetros.
- *SetMagics(booolean, int[])*: Guarda para el jugador o para el oponente, dependiendo del booleano recibido por parámetro, un array que representa las posibles magias que puede utilizar, indicando mediante un 1 que tiene suficiente maná como para utilizar dicha magia, y con un 0 en caso de que no pudiera.
- *GetIndexBoard(int[]): int*: Devuelve a partir de la acción recibida por parámetro un índice que representa unívocamente a un movimiento entre dos casillas adyacentes del tablero. El valor devuelto por este método se utilizará para obtener el atributo *actionBoard*, asignando valores en el rango [0…4] para acciones correspondientes a las distintas magias y el valor devuelto por este método sumando 5 para el resto de movimientos del tablero.
- *GetBoxesBoard(int): int[]:* Se trata del método inverso al anterior, que devuelve a partir del valor del atributo de *actionBoard* restando 5, un array donde guarda la acción realizada. Permite obtener a partir de dicho índice las dos casillas adyacentes que producen el movimiento.
- *GetIndex2Types(): int:* Devuelve a partir de los tipos/combinaciones de tipos de las casillas obtenidas en un determinado chequeo, guardado en el atributo auxiliar *boxTypesCheck*, un índice que lo representa unívocamente. Un ejemplo simplificado para 3 tipos de casillas sería:

  [1,0,0]→0 [0,1,0]→1 [0,0,1]→2 [1,1,0]→3 [1,0,1]→4 [0,1,1]→5

  El valor devuelto por este método se utilizará para obtener el atributo *action1Exchange,* asignando valores en el rango [0…4] para acciones correspondientes a las distintas magias y el valor devuelto por este método sumando 5 para los distintos posibles tipos/combinación de tipos de la línea/líneas que se han realizado.
- *CheckChange(Point, Point): boolean:* Comprueba si se produce alguna linea al intercambiar las dos casillas de los puntos introducidos por parámetro. Simula la jugada con posibles líneas en cascada y guarda las casillas a intercambiar de la fila más baja para cada tipo/combinación de tipos en *boxesTo1Exchange[,] y boxesTo2Exchange[,].*

- *CheckBreak(): boolean:* Comprueba si existe alguna línea en el tablero tras simular un determinado movimiento y, en caso afirmativo, marca dichas casillas a -1 para representar que están borradas y comprobar si se produciría alguna línea en cascada.
- *FallBoxes():* Desplaza las casillas hacia abajo del tablero quitando aquéllas que han quedado marcadas a -1 representando que han sido borradas. De esta forma todas las casillas borradas quedan en la parte superior del tablero, simulando una caída de casillas.
- *Save1Exchange(Point, Point):* Se encarga de guardar las casillas a intercambiar de la fila más baja para cada tipo/combinación de tipos en *boxesTo1Exchange[,]*. Usado por el método *CheckChange(Point, Point).*
- *Save2Exchange(Point, Point):* Se encarga de guardar las casillas a intercambiar de la fila más baja para cada tipo/combinación de tipos en *boxesTo2Exchange[,]*. Usado por el método *CheckChange(Point, Point).*
- *StoreData()*: Se trata de uno de los métodos principales, que se encarga de generar y guardar todos los datos necesarios para los distintos logs a partir de los datos ya almacenados en esta clase mediante los métodos *SetIntBoard()*,*SetStatusState()* y *SetMagics()*.

  El procedimiento que se realiza consiste en comprobar todos los posibles cambios de casillas, llamando para cada uno de ellos al método *CheckChange(Point, Point)* para detectar si se produce alguna línea y guardará las casillas a intercambiar de la fila más baja para cada tipo/combinaciones en *boxesTo1Exchange[,] y boxesTo2Exchange[,].*
- *StoreAction(int[])*: Se trata de otro de los métodos principales y que, en este caso, a partir de la acción que recibe por parámetro, permite calcular para cada tipo de log el valor de su acción correspondiente, que se almacenará en la variable respectiva: *actionBoard, action1Exchange y action2Exchange*.

  El procedimiento que realiza es muy simple en caso de que el jugador haya realizado una magia, ya que en todos los logs el valor que se almacena siempre es el índice de la magia utilizada. En caso de que se realice un movimiento, para el log *logBoard* se guardará en el atributo *actionLog* el valor devuelto por el método *GetIndexBoard(int[])*, que permite obtener a partir de la acción recibida un índice que representa unívocamente a un movimiento entre dos casillas adyacentes del tablero.

  El caso se complica aún más para los logs *log1Exchange* y *log2Exchange*, ya que será necesario simular el movimiento y realizar las comprobaciones mediante el método *CheckChange(Point, Point)* para obtener los tipos/combinaciones de tipos de la línea/líneas generadas en ese movimiento, y guardar en los atributos *action1Exchange* y *action2Exchange* el valor que se considere mejor de todos los posibles que se produzcan al realizar ese movimiento.

El orden en el que se consideran los mejores movimientos es:

1. Combinaciones de los tipos de ataque y maná.
2. Combinaciones de tipos que implican casillas de ataque.
3. Combinaciones de tipos que implican casillas de maná.
4. Cualquier otra combinación de tipos.
5. Línea simple de tipo ataque.
6. Línea simple de tipo maná.
7. Línea simple de cualquier otro color.
8. Línea simple de tipo basura.

Los 4 primeros casos no se tienen en cuenta para el atributo *action1Exchange* del log *log1Exchange*, ya que este tipo de log sólo evalúa las líneas simples. Estos casos se utilizarán para el atributo *action2Exchange* del log *log2Exchange*.

Los valores de los índices de los atributos correspondientes a las acciones que se guardan en este método se especificarán con más detalle en el apartado del *jugador automático* correspondiente a la *generación de logs*.

- *CheckExistLog() y CreateTemplate()*: Comprueban si existen los correspondientes ficheros físicos de log en la ruta correspondiente y, en caso de que no sea así, los genera estableciendo para cada uno de ellos una plantilla inicial. La plantilla inicial genera una cabecera necesaria para los logs generados para cumplir con el formato de los archivos .arff del programa *Weka*, con el que posteriormente se realizará el aprendizaje automático para obtener conocimiento.
- *SaveLog() y WriteLog(String, String, boolean):* Guardan los diferentes logs de la partida en sus correspondientes ficheros. Para ello, *SaveLog()* obtiene de todos los atributos de esta clase toda la información guardada y la recopila en una única cadena de texto que, posteriormente, será enviada como uno de los parámetros al método *WriteLog()* que se encargará finalmente de realizar la escritura de dichos datos en el fichero. Por cada acción realizada se guardará una nueva línea en cada log, que cumplirá con el formato oportuno para ser interpretado por el programa *Weka*.

Esta clase representa un papel fundamental a la hora de implementar la IA del jugador automático ya que los logs generados serán utilizados posteriormente para obtener conocimiento a partir de técnicas de aprendizaje automático. Por ello, **algunos de los atributos** y métodos comentados anteriormente **se especificarán con más detalle en el apartado del *jugador automático*** correspondiente a la ***generación de logs***.

4.4.8 Módulo de IA

El módulo de IA pertenece a la capa de suplementos y está encargado de establecer el comportamiento de la IA del jugador automático a partir de la información recibida por parte del módulo de juego y permite devolverle al mismo el movimiento a realizar por dicho jugador. Este módulo se corresponde con el diagrama de clases mostrado en la siguiente ilustración:

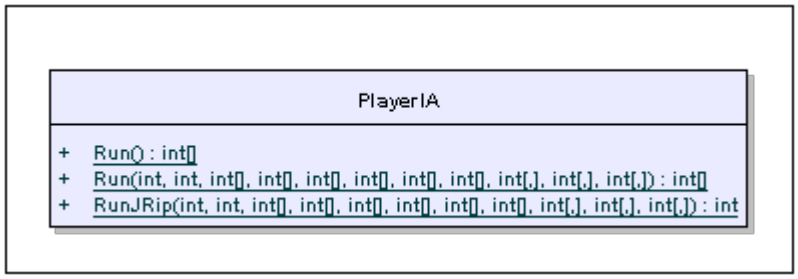

Ilustración 4-11: Módulo de IA

Se trata de un módulo que consta únicamente de la clase estática ***PlayerIA***, cuya finalidad es la de encapsular el comportamiento del jugador automático, estableciendo un método principal llamado *Run()*, que será llamado por el método *UpdateIA(GameTime)* de la clase *GameScreen* del módulo de juego cada vez que se establezca un turno de un jugador automático. El método *Run()* tomará una serie de decisiones en función del estado actual del combate, recibido por parámetro, y devolverá una posible acción a realizar por parte del jugador automático, que será interpretada por el método *UpdateIA(GameTime)* y se comprobará que sea una acción válida. En caso de que no sea así se devuelve de nuevo el control al método *Run()*de este módulo activando que la anterior acción no era válida, para esperar una nueva acción y si finalmente después de varias veces no devuelve una acción permitida, se considera que ha perdido su turno.

El método *Run()* comentado posee sobrecarga, ya que existe una versión simplificada del mismo sin argumentos. Este método actúa mediante prueba y error, devolviendo secuencialmente todas las acciones posibles hasta que una de ellas sea válida. El propio método se desentiende de comprobar si la acción que va a devolver es correcta, ya que aprovecha que se lleva a cabo un control en el método que realiza la llamada, *UpdateIA(GameTime)* de la clase *GameScreen* del módulo de juego.

La otra sobrecarga del método *Run()* recibe una gran cantidad de parámetros que definen completamente el estado del combate. Es utilizada actualmente para emular al jugador automático, devolviendo en cada momento la acción que considera más apropiada. Su comportamiento está basado en las reglas de decisión establecidas en el método *RunJRip()*, las cuales han sido generadas por el algoritmo *JRip* del programa de aprendizaje automático *Weka* a partir de los datos obtenidos con el módulo de log descrito en el punto anterior.

Los parámetros que recibe este método son los siguientes:

- *count*: Número de intento de movimiento para el jugador automático del turno actual [0… n]. Su valor habitual debería ser 0, ya que valores superiores a 0 indican que el método ya ha sido llamado anteriormente y la acción que devolvió no era correcta. Este valor permite la posibilidad de establecer un nuevo mecanismo de selección de acción en caso de que el principal haya devuelto una jugada incorrecta por alguna causa.
- *currentTurn*: Indica el turno actual [1 ó 2]. Permite la posibilidad de establecer distintos comportamientos del jugador automático para cada uno de los dos jugadores.
- *statusPlayer[] y statusOpponent[]*: Estatus del jugador del turno actual y del oponente a éste, respectivamente. Los atributos que se guardan en dicho estatus son: vida, maná, rojo, amarillo, verde y azul.
- *statePlayer[] y stateOpponent[]:* Estados en los que se encuentra el jugador del turno actual y el oponente de este, respectivamente. Para el índice 0 guarda si el jugador tiene el bloqueo de escudo activado, y para el índice 1 guarda si el jugador está envenenado. Se indica con un 1 en caso de que esté activo dicho estado y con un 0 en caso contrario.
- *magicPlayer[] y magicOpponent[]:* Arrays que indican mediante un 1 que el jugador o el oponente respectivamente, tiene suficiente maná como para utilizar dicha magia, y con un 0 en caso de que no pudiera.
- *boxesTo1Exchange[,] y boxesTo2Exchange[,]:* Arrays que guardan si se puede obtener un determinado tipo/combinación de tipos, indicándolo con un 0 en caso negativo, con un 1 en caso de que se pueda realizar una línea de 3 casillas, y con un 2 en caso de que se pueda realizar una línea de 4 ó 5 casillas. Si dicho valor es un 1 ó 2 se guardarán también cuales son las casillas de la fila más baja posible a intercambiar para obtener dicho tipo/combinación de tipos.
- *intBoard[,]*: Representación numérica del tablero. La correspondencia es la siguiente: ataque → 0, maná → 1, rojo → 2, amarillo → 3, verde → 4, azul → 5 y basura → 6.

El valor devuelto encapsula la acción a realizar, y debe cumplir la siguiente codificación para que pueda ser interpretado por el método *UpdateIA(GameTime)* de la clase *GameScreen* del módulo de juego que ha realizado la llamada:

- *action[0]*: Contiene información de control:
  - *action[0]* = 0 → Realizar una magia.
    - *action [1]* = Índice de la magia a realizar.
  - *action[0]* = 1 ó 2 → Realizar un movimiento en el tablero.
    - *action [1-4]* → Coordenadas [x,y] de las 2 casillas a intercambiar.

4.5 Jugador automático

El desarrollo del jugador automático abarca todo el proceso llevado a cabo para dotar de un comportamiento inteligente a los jugadores controlados por el sistema. Se ha establecido un apartado propio debido a que se trata de un **proceso de experimentación** extenso, en el que se han utilizado **técnicas de aprendizaje automático y de IA** para alcanzar el objetivo deseado. Por todo ello, este apartado va más allá de la descripción realizada en los módulos de log y de IA del apartado de *diseño detallado*, definiendo todo el proceso de experimentación que ha sido llevado a cabo para el **desarrollo del jugador automático**.

El proceso de selección de la técnica concreta de Inteligencia Artificial a utilizar y de la herramienta usada para ello se ha realizado anteriormente en el apartado *2.4 Inteligencia Artificial en los videojuegos*, habiendo establecido como **técnica *'árboles y reglas de decisión'*** y como **herramienta *Weka***. En este apartado se parte de dichas premisas, describiendo los pasos necesarios para el desarrollo de dicha técnica con dicha herramienta.

En primer lugar se llevará a cabo una **especificación detallada de los 3 tipos de logs existentes**, definiendo todos los campos de los que están formados, y sus posibles valores; y exponiendo las razones por las que se ha considerado interesante incluir dicho log. Posteriormente se realizará una **selección de los distintos clasificadores** que con los que se van a llevar a cabo los distintos experimentos, estableciendo las características de los mismos y las razones por las que se ha tomado la decisión de usar dichos clasificadores.

Una vez establecidos los distintos tipos de logs y clasificadores se procede a la **descripción de los experimentos realizados**, mostrando los resultados de los mismos y las decisiones tomadas a partir de ellos, que dan lugar a nuevos experimentos. Finalizada esta fase se obtendrá una salida que, adaptada al sistema, permitirá simular la IA del jugador automático.

4.5.1 Definición de los logs

La generación de logs es necesaria para su posterior uso con técnicas de aprendizaje automático para el desarrollo de la IA del jugador automático. Dicha generación se lleva a cabo y ha sido descrita anteriormente en el módulo de log, creando **3 tipos de logs** diferentes que guardan, con distintos matices, el estado del combate y la acción llevada a cabo en cada turno. Estos datos serán guardados en la ruta "*MagicQuest\bin\x86\Debug\Content\logs*" en archivos con extensión .arff con los siguientes nombres: *logBoard*, *log1Exchange* y *log2Exchange.*

Los logs generados se guardan en **formato .arff**, cumpliendo con todas las características de los mismos, debido a que se trata del formato que interpreta el programa de aprendizaje automático que se utilizará, *Weka*. De esta forma no es necesario realizar ninguna transformación posterior a los datos para poder comenzar a trabajar con ellos.

Como en cualquier experimento de inteligencia artificial, es **muy importante determinar los datos que se van a guardar y la representación de los mismos**, ya que tienen una gran influencia en los resultados obtenidos. Debido a que a priori no se puede saber concretamente qué datos y qué forma de guardarlos generarán mejores resultados, se ha decidido crear 3 tipos de logs que guarden el estado del combate con distintos matices.

La **información guardada** en todos los logs refleja el estado del combate y la acción realizada **para ambos jugadores**, diferenciando a cada uno de ellos por el valor del atributo guardado *turn-player-or-opponent.* Por lo tanto si se quiere realizar un estudio del comportamiento de un solo jugador se debe realizar un filtro por el valor de ese atributo seleccionando únicamente uno de los dos jugadores.

En lo **referente al turno y a los distintos jugadores** se guardan **los mismos datos para todos los logs** existentes, ya que no se ha considerado que existan distintas posibilidades a la hora de representar los datos. Por lo tanto, para los 3 logs posibles se guardan los siguientes atributos en el archivo .arff:

- *turn-player-or-opponent numeric*: Indica con un valor numérico a qué jugador pertenece el turno: 1 → Turno jugador 1; 2 → Turno jugador 2.
- *player-health numeric:* Cantidad de vida del jugador del turno actual.
- *player-mana numeric*: Cantidad de maná del jugador del turno actual.
- *player-red numeric*: Cantidad de rojo del jugador del turno actual.
- *player-yellow numeric*: Cantidad de amarillo del jugador del turno actual.
- *player-green numeric*: Cantidad de verde del jugador del turno actual.
- *player-blue numeric*: Cantidad de azul del jugador del turno actual.
- *player-have-shield {0, 1}:* Indica si el jugador del turno actual tiene activado el bloqueo de escudo. 0 → No; 1 → Sí.
- *player-is-poisoned {0, 1}:* Indica si el jugador del turno actual está envenenado. 0 → No; 1 → Sí.
- *player-magic-0 {0, 1}:* Indica si el jugador del turno actual puede realizar la magia '*Curación*'. 0 → No puede; 1 → Sí puede.
- *player-magic-1 {0, 1}:* Indica si el jugador del turno actual puede realizar la magia '*Bloqueo de escudo'*. 0 → No puede; 1 → Sí puede.
- *player-magic-2 {0, 1}:* Indica si el jugador del turno actual puede realizar la magia '*Envenenar'*. 0 → No puede; 1 → Sí puede.
- *player-magic-3 {0, 1}:* Indica si el jugador del turno actual puede realizar la magia '*Golpe ligero'*. 0 → No puede; 1 → Sí puede.
- *player-magic-4 {0, 1}:* Indica si el jugador del turno actual puede realizar la magia '*Golpe crítico'*. 0 → No puede; 1 → Sí puede.

- *opponent-health numeric:* Cantidad de vida del oponente del jugador del turno actual.
- *opponent-mana numeric*: Cantidad de maná del oponente del jugador del turno actual.
- *opponent-red numeric*: Cantidad de rojo del oponente del jugador del turno actual.
- *opponent-yellow numeric*: Cantidad de amarillo del oponente del jugador del turno actual.
- *opponent-green numeric*: Cantidad de verde del oponente del jugador del turno actual.
- *opponent-blue numeric*: Cantidad de azul del oponente del jugador del turno actual.
- *opponent-have-shield {0, 1}:* Indica si el oponente del jugador del turno actual tiene activado el bloqueo de escudo. 0 → No; 1 → Sí.
- *opponent-is-poisoned {0, 1}:* Indica si el oponente del jugador del turno actual está envenenado. 0 → No; 1 → Sí.
- *opponent-magic-0 {0, 1}:* Indica si el oponente del jugador del turno actual puede realizar la magia '*Curación*'. 0 → No; 1 → Sí.
- *opponent-magic-1 {0, 1}:* Indica si el oponente del jugador del turno actual puede realizar la magia '*Bloqueo de escudo'*. 0 → No; 1 → Sí.
- *opponent-magic-2 {0, 1}:* Indica si el oponente del jugador del turno actual puede realizar la magia '*Envenenar'*. 0 → No; 1 → Sí.
- *opponent-magic-3 {0, 1}:* Indica si el oponente del jugador del turno actual puede realizar la magia '*Golpe ligero'*. 0 → No; 1 → Sí.
- *opponent-magic-4 {0, 1}:* Indica si el oponente del jugador del turno actual puede realizar la magia '*Golpe crítico'*. 0 → No; 1 → Sí.

Es la **forma de representar el tablero** y, por tanto, también la **acción realizada**, las que **varían para cada uno de los posibles logs**. El objetivo de cada una de dichas representaciones, así como los atributos necesarios para las mismas, se explicará a continuación para cada tipo log.

El log ***logBoard*** pretende guardar el estado del tablero sin realizar ninguna simplificación del mismo. Para ello guardará en un atributo independiente el valor en formato numérico para cada una de las 64 casillas que componen el tablero, identificando cada tipo de casilla con un determinado valor numérico. En lo que respecta a la acción realizada por el jugador, en este tipo de log se guardará el valor que contiene el atributo *actionBoard* de la clase *log*, que identifica las distintas magias que puede realizar el jugador, y los distintos pares de casillas que se pueden intercambiar en el tablero unívocamente.

Los atributos específicos para este tipo de log concreto son los siguientes:

- *board-box-0 – board-box-63 - {0, 1, 2, 3, 4, 5, 6}*: Representación numérica del tablero para cada una de las 64 casillas. La correspondencia es la siguiente: ataque → 0, maná → 1, rojo → 2, amarillo → 3, verde → 4, azul → 5 y basura → 6.
- *action-board numeric:* Valor que representa unívocamente todas las posibles acciones que se pueden realizar en el combate para este tipo de log en concreto. Los valores que puede tomar son los siguientes:
    - *[0 - 4]:* Indica que se ha realizado una determinada magia:

      0 → Curación; 1 → Bloqueo de escudo; 2 → Curación; 3 → Golpe ligero; 4 → Golpe crítico.
    - *[5 - 60]:* Indica que se han intercambiado dos casillas las cuales se encuentran situadas una encima de la otra. El valor más bajo corresponde con un intercambio entre la casilla de la esquina inferior izquierda y la superior a esta. Se va incrementando el valor para las casillas situadas más a la derecha de esa misma fila y los siguientes valores se corresponden a los de las filas superiores, correspondiendo el valor más alto para el cambio que involucra a la casilla situada en la esquina superior derecha.
    - *[61 - 116]:* Indica que se han intercambiado dos casillas las cuales se encuentran situadas una al lado de la otra. La numeración sigue el mismo procedimiento que en el caso anterior, correspondiendo el valor más bajo a aquel que involucra a la casilla situada en la esquina inferior izquierda, y el más alto al que involucra a la casilla de la esquina superior derecha.

Los **resultados esperados** para este log son poco halagüeños, debido al elevado número de atributos existentes y la cantidad de posibles valores que poseen los mismos, que provocan que el espacio de búsqueda sea inmensamente elevado, existiendo demasiadas combinaciones de valores posibles como para que se pueda obtener algún patrón o conocimiento de los datos. Además, el número de acciones posibles también es elevado, dificultando la clasificación de las mismas.

El log ***log1Exchange*** pretende simplificar la representación del tablero, guardando únicamente los datos referentes a los posibles tipos de casillas de los que se pueden hacer línea para la situación concreta del tablero. De esta forma se pretende reducir el espacio de búsqueda, esperando obtener mejores resultados que en el caso anterior.

En lo que respecta a la acción realizada por el jugador también se simplifica, reduciendo su número de forma que identifiquen las distintas magias que puede realizar el jugador y los distintos tipos de líneas que se pueden obtener en el tablero, de forma unívoca. El valor de este acción será aquel que está guardado en el atributo *action1Exchange* de la clase *log*.

Los atributos específicos para este tipo de log concreto son los siguientes:

- *can-get-attack {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo ataque. Puede tomar los siguientes valores:
  - *0* → No existe línea.
  - *1* → Existe línea de 3 casillas.
  - *2* → Existe línea de 4 ó 5 casillas.
- *can-get-mana {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo maná. Tomará los mismos valores que los anteriores.
- *can-get-red {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo rojo. Tomará los mismos valores que los anteriores.
- *can-get*-yellow *{0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo amarillo. Tomará los mismos valores que los anteriores.
- *can-get-green {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo verde. Tomará los mismos valores que los anteriores.
- *can-get-blue {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo azul. Tomará los mismos valores que los anteriores.
- *can-get-waste {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo basura. Tomará los mismos valores que los anteriores.
- *action-1Exchange numeric:* Valor que representa unívocamente todas las posibles acciones que se pueden realizar en el combate para este tipo de log en concreto. Los valores que puede tomar son los siguientes:
  - *[0 - 4]:* Indica que se ha realizado una determinada magia:

    0 → Curación; 1 → Bloqueo de escudo; 2 → Curación; 3 → Golpe ligero; 4 → Golpe crítico.
  - *[5 - 11]:* Indica el tipo de línea generada con el movimiento que se ha realizado:

    5 → Ataque; 6 → Maná; 7 → Rojo; 8 → Amarillo; 9 → Verde; 10 → Azul; 11 → Basura.

Los **resultados esperados** para este log deberían ser relativamente buenos ya que se han reducido el número de atributos a 34, disminuyendo el número de combinaciones posibles en el espacio de búsqueda y además la información guardada aporta mucho más significado que en el caso anterior, ya que quizá no sea tan importante las casillas concretas a intercambiar sino que seguramente sea más importante el fin por el cual se intercambian dichas casillas, es decir, el tipo de línea que se realiza con ese movimiento. En lo que respecta al número de acciones posibles también se han reducido sustancialmente, pasando de 116 posibles valores a tan solo 11, con lo cual es probable que resulte más sencillo clasificar los datos. No obstante, se corre el peligro de haber simplificado demasiado los datos representados y que se produzca una perdida de información demasiado elevada.

Por último el log ***log2Exchange*** pretende continuar con el sistema definido en el punto anterior pero guardando datos más elaborados de forma que se aporte más complejidad e información. En este caso no solo se guardarán los datos referentes a los posibles tipos de casillas que se pueden hacer línea para una situación concreta del tablero, sino que se irá más allá y se simulará el estado en el que quedaría el tablero después de la misma, comprobando si se produciría una nueva línea. De esta forma ya no se obtendrán únicamente los tipos de casillas que forman línea sino también combinaciones de tipos, en el caso de que se produjera alguna línea en cascada.

De esta forma se pretende mejorar el log anterior, que quizá tenía un planteamiento demasiado simple, para dotar de nueva información que anteriormente no quedaba reflejada. En este sentido, por ejemplo, puede ocurrir en un caso en que el jugador necesite líneas amarillas, se decante por realizar una línea roja que a su vez produce en cascada la necesaria línea amarilla. Con el log del punto anterior únicamente quedaba reflejado que el jugador realizaba una línea roja, información que nada tenía que ver con el objetivo del jugador, mientras que en este caso queda reflejado que realiza una combinación de líneas roja-amarilla, tratándose de un movimiento bastante bueno.

Los atributos específicos para este tipo de log concreto son, además de los especificados para el log *log1Exchange* a excepción de la acción del mismo, los siguientes:

- *can-get-attack-mana {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea de ataque y otra de maná.
- *can-get-attack-red {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea de ataque y otra roja.
- *can-get-attack-yellow {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea de ataque y otra amarilla.
- *can-get-attack-green {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea de ataque y otra verde.
- *can-get-attack-blue {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea de ataque y otra azul.
- *can-get-mana-red {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea de maná y otra roja.
- *can-get-mana-yellow {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea de maná y otra amarilla.
- *can-get-mana-green {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea de maná y otra verde.
- *can-get-mana-blue {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea de maná y otra azul.

- *can-get-red-yellow {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea roja y otra amarilla.
- *can-get-red-green {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea roja y otra verde.
- *can-get-red-blue {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea roja y otra azul.
- *can-get-yellow-green {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea amarilla y otra verde.
- *can-get-yellow-blue {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea amarilla y otra azul.
- *can-get-green-blue {0, 1, 2}:* Indica si se puede realizar algún movimiento en el que se obtenga una línea verde y otra azul.
- *action-2Exchange numeric:* Valor que representa unívocamente todas las posibles acciones que se pueden realizar en el combate para este tipo de log en concreto. Los valores que puede tomar son los siguientes:
    - *[0 - 4]:* Indica que se ha realizado una determinada magia:

      0 → Curación; 1 → Bloqueo de escudo; 2 → Curación; 3 → Golpe ligero; 4 → Golpe crítico.
    - *[5 - 11]:* Indica que el jugador ha realizado un movimiento en el que se obtiene una única línea del siguiente tipo:

      5 → Ataque; 6 → Maná; 7 → Rojo; 8 → Amarillo; 9 → Verde; 10 → Azul; 11 → Basura.
    - *[12 - 26]:* Indica que el jugador ha realizado un movimiento en el que se obtiene una combinación de líneas de los siguientes tipos:

      12 → Ataque/Maná; 13 → Ataque/Rojo; 14 → Ataque/Amarillo; 15 → Ataque/Verde; 16 → Ataque/Azul; 17 → Maná/Rojo; 18 → Maná/Amarillo; 19 → Maná/Verde; 20 → Maná/Azul; 21 → Rojo/Amarillo; 22 → Rojo/Verde; 23 → Rojo/Azul; 24 → Amarillo/Verde; 25 → Amarillo/Azul; 26 → Verde/Azul.

Los **resultados esperados** para este log deberían ser los mejores de entre los 3 posibles logs existentes, ya que aporta mayor información que en el caso anterior pero sin que ello implique un gran aumento de la complejidad y del espacio de búsqueda.

En este caso se ha incrementado el número de atributos a 50 y existen 26 posibles acciones con respecto a 34 y 11 del log anterior respectivamente, hecho que se considera positivo ya que no se trata de un número demasiado elevado y sin embargo permite establecer una clasificación más detallada, eliminando la posible sencillez existente en el caso anterior.

4.5.2 Selección de clasificadores

***Weka* permite el uso de una gran cantidad de clasificadores**, agrupados por categorías. Debido a las características del jugador automático que se quiere construir, se ha considerado oportuno que la técnica utilizada esté basada en **árboles o reglas de decisión**, debido a que la salida generada por las mismas es más sencilla que la de otras técnicas y, por tanto, es más fácil de adaptar y codificar en el método *Run()* de la clase *PlayerIA* del módulo de IA que será la que encapsule el comportamiento del jugador automático en *Magic Quest*.

Se ha considerado interesante realizar los distintos experimentos tanto con una técnica de **árboles de decisión** como con otra técnica de **reglas de decisión**. Del mismo modo **los experimentos se realizarán con todos los logs** generados, de forma que se pueda realizar una comparación de los resultados obtenidos, determinando cuáles son los que más se ajustan al comportamiento deseado.

Para poder utilizar los distintos clasificadores de árboles y de reglas de decisión que proporciona *Weka* ha sido necesario eliminar el atributo que indica el turno, ya que se quiere tener en cuenta indiferentemente las acciones de ambos jugadores, y aplicar un **filtro para transformar de número a nominal los datos** de la salida proporcionada, que en este caso se trata del atributo que determina la acción realizada. De esta forma se pueden aplicar clasificadores más coherentes con respecto a la solución que se está buscando, obteniendo **una acción concreta para unos determinados valores de entrada**, ya que otros clasificadores devuelven una salida de acuerdo con una función matemática, generando números decimales que sería imposible transformar a una acción concreta.

En lo que respecta a las posibles técnicas de **árboles de decisión** que proporciona *Weka* se ha realizado una prueba inicial con distintos tipos de clasificadores, valorando no sólo el porcentaje de clasificaciones correctas de cada uno de ellos, sino también su facilidad de adaptación y codificación en el sistema desarrollado. Valorando dichos factores tras los resultados obtenidos, finalmente se ha decidido que el clasificador de árboles de decisión con el que se realizarán los experimentos es el ***J48***.

Por otra parte, en el caso de las posibles técnicas de **reglas de decisión** también se ha realizado una prueba inicial con distintos tipos de clasificadores, con la misma finalidad que para la técnica anterior. Tras los resultados obtenidos, se ha determinado que el clasificador de reglas de decisión con el que se realizarán los experimentos es el ***JRip***.

Una vez establecido los clasificadores concretos que se van a utilizar para cada una de las técnicas a utilizar, se procederá en el **siguiente punto** a la **explicación detallada de los experimentos** realizados para los distintos tipos de logs, hasta llegar a los resultados finales con los que se ha implementado la inteligencia artificial del jugador automático del juego.

4.5.3 Experimentación con Weka

La experimentación realizada tiene como objetivo dotar de comportamiento inteligente al jugador automático en *Magic Quest.* El procedimiento realizado consta de una **primera fase** en la cual se realiza **una prueba de cada uno de los logs generados con cada uno de los clasificadores** seleccionados para determinar, según los resultados obtenidos, cuál de los 3 tipos de logs existentes proporciona el mayor índice de aciertos, generando mejores resultados.

Existe una **segunda fase** de refinamiento en la cual se intenta mejorar aun más el índice de aciertos para el log que aporte mejores resultados. Para ello, nos centraremos únicamente en dicho tipo de logs y **se realizarán distintos filtros y modificaciones de los atributos**, descartando los menos relevantes y aportando mayor peso a los más relevantes. El objetivo de esta fase consiste afinar el resultado final, intentando mejorar el jugador automático final que se implementará en el juego.

Los distintos logs con los que se realizarán todos los experimentos de este apartado se han generado jugando una gran cantidad de **partidas** diferentes en el **modo *2 jugadores***, controlando las decisiones de ambos jugadores por una única persona, hasta alcanzar un total de **2000 registros** correspondientes a esa misma cantidad de estados en los que se ha ido encontrando el desarrollo del combate, asociados a la correspondiente acción que ha realizado el jugador en ese caso.

- **Fase 1: Comparación de logs**

Esta primera fase pretende realizar un **estudio entre los distintos tipos de logs** generados con el objetivo de determinar cual de ellos proporciona, con respecto a los clasificadores seleccionados, una serie de decisiones que clasifiquen los registros guardados con el mayor índice de aciertos posibles respecto a las acciones que había realizado el jugador para ese mismo caso. Los experimentos realizados en esta fase y sus resultados correspondientes son los siguientes:

- ***Experimentos 1a y 1b: logBoard***

El resultado de aplicar el clasificador *JRip* al log *logBoard* es el siguiente:

Number of Rules:	78	TOTAL	%
Correctly Classified Instances		206	10.3 %
Incorrectly Classified Instances		1794	89.7 %
Total Number of Instances		2000	

Tabla 4-1: Fase 1 - Experimento 1a: logBoard - JRip

El resultado de aplicar el clasificador *J48* al log *logBoard* es el siguiente:

Size of the tree:	1942	**TOTAL**	**%**
Correctly Classified Instances		72	3.6 %
Incorrectly Classified Instances		1928	96.4 %
Total Number of Instances		2000	

Tabla 4-2: Fase 1 - Experimento 1b: logBoard – J48

Tal y como se esperaba para este tipo de logs, los resultados son realmente malos, obteniéndose únicamente unos porcentajes de aciertos de un 10,3% y un 3,6% respecto a las acciones realizadas por el jugador, para los clasificadores *JRip* y *J48* respectivamente. Las causas de estos resultados negativos se deben al elevado número de atributos existentes y la cantidad de posibles valores que poseen los mismos, que provocan que el espacio de búsqueda sea muy amplio, existiendo demasiadas combinaciones de valores posibles como para que se pueda obtener algún patrón o conocimiento de los datos. Además, el número de acciones posibles también es elevado, dificultando la clasificación de las mismas.

Tras estos resultados, los cuales eran esperados, se puede llegar a la conclusión de que este tipo de log no es útil para obtener el comportamiento del jugador automático, debido a que sus resultados son muy negativos. Se espera que los otros dos tipos de logs aporten mejores resultados.

- ***Experimentos 2a y 2b: log1Exchange***

Los resultados de los distintos experimentos para el log *log1Exchange* son:

Number of Rules:	34	**TOTAL**	**%**
Correctly Classified Instances		1734	86.7 %
Incorrectly Classified Instances		266	13.3 %
Total Number of Instances		2000	

Tabla 4-3: Fase 1 - Experimento 2a: log1Exchange - JRip

Size of the tree:	211	**TOTAL**	**%**
Correctly Classified Instances		1792	89.6 %
Incorrectly Classified Instances		208	10.4 %
Total Number of Instances		2000	

Tabla 4-4: Fase 1 - Experimento 2b: log1Exchange – J48

Los resultados obtenidos en estos experimentos son bastante mejores que los de los casos anteriores, tal y como se esperaba, obteniéndose para el clasificador *JRip* un 86,7% de aciertos con tan sólo 34 reglas de decisión y para el clasificador J48 aún mejores resultados con un 89,6% de aciertos y un árbol de decisión de 211 nodos de tamaño.

La razón de esta mejora se debe que se ha producido una reducción considerable en el número de combinaciones posibles en el espacio de búsqueda y además la información guardada aporta mayor significado, identificando en este caso el tipo de línea que se realiza con el movimiento, en lugar de las casillas intercambiadas. Respecto a la acción realizada también se han reducido sustancialmente, pasando de 116 posibles valores a tan solo 11 diferentes, con lo cual es probable que resulte más sencillo clasificar los datos.

Sin embargo existe la incertidumbre de que se haya podido simplificar demasiado los datos representados, provocando una pérdida demasiado elevada de información, por lo que se piensa que los siguientes experimentos, que se realizarán con el log *log2Exchange*, proporcionarán resultados aún mejores que estos, al ampliar la información aportada.

- ***Experimentos 3a y 3b: log2Exchange***

Por último, al aplicar el clasificador *JRip* al log *log2Exchange* se obtiene el siguiente resultado:

Number of Rules:	65	**TOTAL**	**%**
Correctly Classified Instances		1671	83.55 %
Incorrectly Classified Instances		329	16.45 %
Total Number of Instances		2000	

Tabla 4-5: Fase 1 - Experimento 3a: log2Exchange - JRip

Para el caso del clasificador *J48* con respecto al log *log2Exchange* el resultado es el siguiente:

Size of the tree:	309	**TOTAL**	**%**
Correctly Classified Instances		1730	86.5 %
Incorrectly Classified Instances		270	13.5 %
Total Number of Instances		2000	

Tabla 4-6: Fase 1 - Experimento 3b: log2Exchange – J48

En este último caso, que se esperaba que obtuviera los mejores resultados, la experimentación realizada demuestra que no es así, ya el porcentaje de ciertos para el clasificador *JRip* ha bajado al 83,55% respecto al 86,7% del caso anterior, aumentando además el número de reglas de 34 a 65. En lo que respecta al clasificador *J48* tampoco mejora los resultados, ya que se obtiene un porcentaje de aciertos del 86,5% con respecto al 89,6% de caso anterior, generando además un árbol de 309 nodos, más grande que en el caso anterior, que era de 211.

La conclusión que se obtiene tras obtener estos resultados es que a pesar de que este tipo de log contiene una mayor información acerca del estado del combate, la complejidad añadida que posee para incorporar dicha información hace que aumente demasiado el espacio de búsqueda y la complejidad del problema, obteniendo soluciones más extensas que en el caso anterior, con un porcentaje de aciertos menor.

Por lo tanto, el tipo de log con el que se realizará la fase 2 de experimentos será *log1Exchange*, ya que no solo consigue un porcentaje de aciertos mayor que los otros, sino también que las soluciones generadas sean más sencillas, facilitando su implementación a la hora de introducirla en el jugador automático del juego.

- **Fase 2: Refinamiento**

Esta segunda fase pretende **mejorar aun más el índice de aciertos** para el log *log1Exchange*, que ha aportado los mejores resultados de los 3 logs existentes. Para ello se realizarán con dicho log una serie de experimentos para cada uno de los clasificadores, aplicando una serie de filtros y modificaciones de los atributos, intentando descartar los menos relevantes y aportando mayor peso a los más relevantes. Los experimentos realizados en esta fase y sus resultados correspondientes son los siguientes:

- ***Experimentos 1a y 1b: Logs iniciales***

El primer experimento de esta fase vuelve a ser el mismo que el realizado en la primera fase para el log *log1Exchange*, de forma que se obtienen los mismos resultados, que son los siguientes para los 2 clasificadores existentes:

Number of Rules:	34	**TOTAL**	**%**
Correctly Classified Instances		1734	86.7 %
Incorrectly Classified Instances		266	13.3 %
Total Number of Instances		2000	

Tabla 4-7: Fase 2 - Experimento 1a: JRip – Inicial

Size of the tree:	211	TOTAL	%
Correctly Classified Instances		1792	89.6 %
Incorrectly Classified Instances		208	10.4 %
Total Number of Instances		2000	

Tabla 4-8: Fase 2 - Experimento 1b: J48 - Inicial

La razón por la que se vuelven a mostrar estos resultados es para tenerlos presentes a la hora de compararlos con los de futuros experimentos y poder detectar si realmente se produce alguna mejoría en los mismos con las modificaciones que se irán introduciendo.

- ***Experimentos 2a y 2b: Atributos vida - mana***

La primera modificación que se realizará a los atributos del log tiene como objetivo facilitar la toma de decisión del jugador a la hora de usar la magia 4, '*Golpe crítico'*, que consiste en restar al jugador contrario una cantidad de vida igual al maná dorado acumulado, tratándose de una magia muy determinante a la hora de decidir el desenlace del combate.

Debido a que se trata de una magia que requiere bastante maná de distintos colores para ser usada, cualquier jugador humano intentaría realizarla cuando el maná dorado acumulado sea igual o mayor que la vida del enemigo, o por lo menos, cuando se tenga una cantidad bastante alta. Del mismo modo, en el caso de que el jugador contrario pueda realizarla y tenga una gran cantidad de maná dorado acumulado, un jugador humano intentaría protegerse contra la misma utilizando la magia 0, '*Curación'*, para incrementar la vida y evitar morir con dicho golpe, o bien la magia 1, '*Bloqueo de escudo'*, que reduce a 0 el daño recibido en el siguiente golpe y así evitaría en gran daño que produce.

Con los datos guardados en el log actual, el clasificador no consigue obtener dicha información al ser demasiado compleja la relación entre los datos existentes. Por ello se ha decidido introducir dos nuevos atributos calculados a partir de los anteriores, que aportan información adicional que los jugadores humanos tenían en cuenta a la hora de tomar las decisiones de sus acciones, y sin embargo actualmente el clasificador no tenía forma de determinarlo. Los nuevos atributos incrementan el total a 36, y son calculados de la siguiente forma:

- *player-healt-minus-opp-mana numeric*: Vida del jugador menos maná dorado del oponente. Valores iguales o inferiores a 0 indican peligro, ya que el oponente podría matar al jugador con la magia 4, *'Golpe crítico'*.
- *opp-healt-minus-player-mana numeric*: Vida del oponente menos maná dorado del jugador. Valores iguales o inferiores a 0 indican que se puede matar al oponente con la magia 4, *'Golpe crítico'*.

Los resultados al aplicar el clasificador *JRip* tras introducir dichas variables son los siguientes:

Number of Rules:	34	**TOTAL**	**%**
Correctly Classified Instances		1753	87.65 %
Incorrectly Classified Instances		247	12.35 %
Total Number of Instances		2000	

Tabla 4-9: Fase 2 - Experimento 2a: JRip – Vida menos Maná

Para el caso del clasificador *J48* al introducir dichas variables se obtienen estos nuevos resultados:

Size of the tree:	214	**TOTAL**	**%**
Correctly Classified Instances		1790	89.5 %
Incorrectly Classified Instances		210	10.5 %
Total Number of Instances		2000	

Tabla 4-10: Fase 2 - Experimento 2b: J48 – Vida menos Maná

Los **resultados** obtenidos en lo que respecta al porcentaje de instancias correctamente clasificadas se ha mantenido inalterado, obteniendo no solo porcentajes similares, sino también el mismo número de reglas y prácticamente el mismo tamaño del árbol. No obstante, hay que fijarse con más detalle en los resultados que proporciona *Weka*, concretamente en la *matriz de confusión*[22] generada. Para el experimento anterior teníamos que el número de instancias clasificadas correctamente para la magia 4, *'Golpe crítico'*, era únicamente de 9. Tras introducir los nuevos atributos en este experimento las clasificaciones correctas de dicha magia han aumentado hasta 17. Además, como ya se ha comentado, no solo afecta a esta magia, sino también ha influido muy positivamente a la magia 1, *'Bloqueo de escudo'*, ya que ahora es más sencillo determinar cuando usarla, mejorando el índice de aciertos para esta de 12 a 20.

Por lo tanto debido a que se considera que un buen uso de las magias es vital para un buen comportamiento del jugador automático, y esta modificación mejora el porcentaje de aciertos de varias, se considera un experimento positivo, por lo que se continuará la investigación con las modificaciones actuales.

[22] La **Matriz de Confusión** es una tabla de tamaño n*n, siendo n el número de clases donde las columnas indican las categorías clasificadas por el clasificador y las filas las categorías reales de los datos. Por lo que los elementos en la diagonal principal son los elementos que ha acertado el clasificador y lo demás son los errores.

- ***Experimentos 3a y 3b: Eliminando las magias***

La siguiente modificación que se realiza se basa en la eliminación de posibles datos redundantes con la finalidad de reducir el espacio de búsqueda y, de esta forma, intentar mejorar los resultados obtenidos. Se puede observar que los atributos que indican que se puede realizar cada una de las magias, tanto para el jugador como para el oponente, no tienen porqué ser realmente necesarios ya que la información que aportan ya está implícita en la cantidad de maná acumulado. Por ejemplo, no sería necesario saber si se puede utilizar la magia curación, ya que sería equivalente a saber si se tiene 10 de maná azul o más, que es su coste.

De esta forma se eliminan un total de 10 atributos, que se corresponden con las 5 magias de cada uno de los jugadores, reduciendo el número total a 26, que son los siguientes:

- *player-health numeric:* Cantidad de vida del jugador del turno actual.
- *player-mana numeric*: Cantidad de maná del jugador del turno actual.
- *player-red numeric*: Cantidad de rojo del jugador del turno actual.
- *player-yellow numeric*: Cantidad de amarillo del jugador del turno actual.
- *player-green numeric*: Cantidad de verde del jugador del turno actual.
- *player-blue numeric*: Cantidad de azul del jugador del turno actual.
- *player-have-shield {0, 1}:* Indica si el jugador del turno actual tiene activado el bloqueo de escudo. 0 → No; 1 → Sí.
- *player-is-poisoned {0, 1}:* Indica si el jugador del turno actual está envenenado. 0 → No; 1 → Sí.
- *opponent-health numeric:* Cantidad de vida del oponente del jugador del turno actual.
- *opponent-mana numeric*: Cantidad de maná del oponente del jugador del turno actual.
- *opponent-red numeric*: Cantidad de rojo del oponente del jugador del turno actual.
- *opponent-yellow numeric*: Cantidad de amarillo del oponente del jugador del turno actual.
- *opponent-green numeric*: Cantidad de verde del oponente del jugador del turno actual.
- *opponent-blue numeric*: Cantidad de azul del oponente del jugador del turno actual.
- *opponent-have-shield {0, 1}:* Indica si el oponente del jugador del turno actual tiene activado el bloqueo de escudo. 0 → No; 1 → Sí.
- *opponent-is-poisoned {0, 1}:* Indica si el oponente del jugador del turno actual está envenenado. 0 → No; 1 → Sí.

- *can-get-attack {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo ataque. Puede tomar los siguientes valores:
  - *0* → No existe línea.
  - *1* → Existe línea de 3 casillas.
  - *2* → Existe línea de 4 ó 5 casillas.
- *can-get-mana {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo maná. Tomará los mismos valores que los anteriores.
- *can-get-red {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo rojo. Tomará los mismos valores que los anteriores.
- *can-get*-yellow *{0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo amarillo. Tomará los mismos valores que los anteriores.
- *can-get-green {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo verde. Tomará los mismos valores que los anteriores.
- *can-get-blue {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo azul. Tomará los mismos valores que los anteriores.
- *can-get-waste {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo basura. Tomará los mismos valores que los anteriores.
- *player-healt-minus-opp-mana numeric*: Vida del jugador menos mana del oponente. Este atributo ha sido calculado en el experimento 1 de esta segunda fase.
- *opp-healt-minus-player-mana numeric*: Vida del oponente menos mana del jugador. Este atributo ha sido calculado en el experimento 1 de esta segunda fase.
- *action-1Exchange numeric:* Valor que representa unívocamente todas las posibles acciones que se pueden realizar en el combate para este tipo de log en concreto.

Los resultados obtenidos al aplicar el clasificador *JRip* tras reducir el número de variables, simplificando el espacio de búsqueda, son los siguientes:

Number of Rules:	35	**TOTAL**	**%**
Correctly Classified Instances		1736	86.8 %
Incorrectly Classified Instances		264	13.2 %
Total Number of Instances		2000	

Tabla 4-11: Fase 2 - Experimento 3a: JRip – Eliminando las magias

Los resultados obtenidos al aplicar el clasificador *J48* a esos mismos datos son los siguientes:

Size of the tree:	211	**TOTAL**	**%**
Correctly Classified Instances		1788	89.4 %
Incorrectly Classified Instances		212	10.6 %
Total Number of Instances		2000	

Tabla 4-12: Fase 2 - Experimento 3b: J48 – Eliminando las magias

Los **resultados** obtenidos en estos experimentos no se han considerado de interés ya que son ligeramente inferiores que en el caso anterior en cuanto a instancias correctamente clasificadas para ambos logs. Pero la verdadera razón por la que se descarta este experimento es porque, al fijarse de nuevo en la *matriz de confusión*, los aciertos obtenidos para todas las magias se reducen sensiblemente, por lo que se considera que un jugador automático generado a partir de estos experimentos tendrá un comportamiento con el uso de las magias menos oportuno que el del caso anterior.

Como conclusión de este experimento fallido se puede sacar que quizá, aunque la información guardada en los distintos atributos sea redundante, tienen distintos matices que son los que marcan la ligera diferencia que existe entre unos u otros resultados.

- ***Experimentos 4a y 4b: Eliminando los manás***

Siguiendo con el razonamiento del experimento anterior, y viendo que eliminando las magias no se obtenían los resultados deseados, se ha probado a realizar justo lo contrario, eliminar los valores de los diferentes manas acumulados por el jugador y por el oponente. De esta forma se realiza justo el procedimiento contrario al del experimento anterior, eliminando de nuevo la redundancia que existe entre los atributos correspondientes a los manas y a las magias.

De nuevo el número de atributos eliminados es 10, que se corresponden al maná dorado, rojo, amarillo, verde y azul de cada uno de los jugadores, reduciendo el número total de atributos a 26, que son los siguientes:

- *player-health numeric:* Cantidad de vida del jugador del turno actual.
- *player-have-shield {0, 1}:* Indica si el jugador del turno actual tiene activado el bloqueo de escudo. 0 → No; 1 → Sí.
- *player-is-poisoned {0, 1}:* Indica si el jugador del turno actual está envenenado. 0 → No; 1 → Sí.

- *player-magic-0 {0, 1}:* Indica si el jugador del turno actual puede realizar la magia '*Curación*'. 0 → No puede; 1 → Sí puede.
- *player-magic-1 {0, 1}:* Indica si el jugador del turno actual puede realizar la magia '*Bloqueo de escudo'*. 0 → No puede; 1 → Sí puede.
- *player-magic-2 {0, 1}:* Indica si el jugador del turno actual puede realizar la magia '*Envenenar'*. 0 → No puede; 1 → Sí puede.
- *player-magic-3 {0, 1}:* Indica si el jugador del turno actual puede realizar la magia '*Golpe ligero'*. 0 → No puede; 1 → Sí puede.
- *player-magic-4 {0, 1}:* Indica si el jugador del turno actual puede realizar la magia '*Golpe crítico'*. 0 → No puede; 1 → Sí puede.
- *opponent-health numeric:* Cantidad de vida del oponente del jugador del turno actual.
- *opponent-have-shield {0, 1}:* Indica si el oponente del jugador del turno actual tiene activado el bloqueo de escudo. 0 → No; 1 → Sí.
- *opponent-is-poisoned {0, 1}:* Indica si el oponente del jugador del turno actual está envenenado. 0 → No; 1 → Sí.
- *opponent-magic-0 {0, 1}:* Indica si el oponente del jugador del turno actual puede realizar la magia '*Curación*'. 0 → No; 1 → Sí.
- *opponent-magic-1 {0, 1}:* Indica si el oponente del jugador del turno actual puede realizar la magia '*Bloqueo de escudo'*. 0 → No; 1 → Sí.
- *opponent-magic-2 {0, 1}:* Indica si el oponente del jugador del turno actual puede realizar la magia '*Envenenar'*. 0 → No; 1 → Sí.
- *opponent-magic-3 {0, 1}:* Indica si el oponente del jugador del turno actual puede realizar la magia '*Golpe ligero'*. 0 → No; 1 → Sí.
- *opponent-magic-4 {0, 1}:* Indica si el oponente del jugador del turno actual puede realizar la magia '*Golpe crítico'*. 0 → No; 1 → Sí.
- *can-get-attack {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo ataque. Puede tomar los siguientes valores:
  - *0* → No existe línea.
  - *1* → Existe línea de 3 casillas.
  - *2* → Existe línea de 4 ó 5 casillas.
- *can-get-mana {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo maná. Tomará los mismos valores que los anteriores.
- *can-get-red {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo rojo. Tomará los mismos valores que los anteriores.
- *can-get-*yellow *{0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo amarillo. Tomará los mismos valores que los anteriores.

- *can-get-green {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo verde. Tomará los mismos valores que los anteriores.
- *can-get-blue {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo azul. Tomará los mismos valores que los anteriores.
- *can-get-waste {0, 1, 2}:* Indica si se puede generar en el tablero una línea de tipo basura. Tomará los mismos valores que los anteriores.
- *player-healt-minus-opp-mana numeric*: Vida del jugador menos mana del oponente. Este atributo ha sido calculado en el experimento 1 de esta segunda fase.
- *opp-healt-minus-player-mana numeric*: Vida del oponente menos mana del jugador. Este atributo ha sido calculado en el experimento 1 de esta segunda fase.
- *action-1Exchange numeric:* Valor que representa unívocamente todas las posibles acciones que se pueden realizar en el combate para este tipo de log en concreto.

Los nuevos resultados obtenidos al descartar los atributos de manas al aplicar los clasificadores *JRip* y *J48* son los siguientes:

Number of Rules:	28	**TOTAL**	**%**
Correctly Classified Instances		1774	88.7 %
Incorrectly Classified Instances		226	11.3 %
Total Number of Instances		2000	

Tabla 4-13: Fase 2 - Experimento 4a: JRip – Eliminando los manas

Size of the tree:	207	**TOTAL**	**%**
Correctly Classified Instances		1799	89.95 %
Incorrectly Classified Instances		201	10.05 %
Total Number of Instances		2000	

Tabla 4-14: Fase 2 - Experimento 4b: J48 – Eliminando los manas

Los nuevos **resultados** tras estas modificaciones han mejorado ligeramente en todos los aspectos, ya que se ha aumentado tanto el índice de aciertos de ambos clasificadores con respecto al experimento 2, y también han aumentado los aciertos obtenidos para todas las magias, mejorando en cierta medida el nivel del comportamiento del jugador automático.

Los resultados de estos dos últimos experimentos muestran que, como se sospechaba, existía redundancia en los datos y que se podían eliminar algunos de los atributos guardados. No obstante, resulta curioso que sea eliminando los valores de maná acumulados, en lugar de las posibles magias que se pueden realizar, lo que obtenga mejores resultados.

La posible razón de este hecho es que las cantidades de maná no aportan mucha más información que los atributos que indican si se puede realizar una determinada magia y, sin embargo, al tratarse estos primeros de atributos numéricos, representan un mayor espacio de búsqueda que los segundos atributos, que únicamente puede tomar 2 valores, por lo que aumenta la complejidad en la clasificación obtenida y empeoran los resultados.

Sin embargo, hay que advertir que eliminar las cantidades de maná puede incurrir en el peligro de que en algún momento, debido a las características del combate, la salida del jugador automático sea adquirir maná de un determinado color, sin poder establecer una cuota máxima y llegando a acumular cantidades elevadas del mismo sin motivo aparente. No obstante, no se considera un inconveniente, ya que los clasificadores han demostrado unos buenos resultados, por lo que las acciones que realizará el jugador automático seguirán asemejándose en gran medida a las realizadas por el jugador con el que se han generado los logs.

- ***Experimentos 5a y 5b: Eliminando magias del oponente***

Este último experimento busca mejorar los resultados obtenidos eliminando una serie de atributos que se ha considerado que pueden no aportar ninguna información. Se persigue de nuevo reducir el espacio de búsqueda, intentando aumentar el índice de aciertos y simplificando las reglas y árboles obtenidos.

Los atributos que se han decidido eliminar son los 4 que indican si el oponente puede realizar una determinada magia, a excepción de la magia 0, '*Curación'*, que sí aporta información ya que permite, por ejemplo, determinar si es buen momento para envenenar al oponente. El resto de atributos no se considera que aporten información ya que un jugador humano no suele tener en cuenta si el oponente puede o no realizar ciertas magias a la hora de tomar la decisión de la acción a realizar. Por lo tanto el número final de atributos es 22, habiendo eliminado los siguientes 4:

- *opponent-magic-1 {0, 1}:* Indica si el oponente del jugador del turno actual puede realizar la magia '*Bloqueo de escudo'*. 0 → No; 1 → Sí.
- *opponent-magic-2 {0, 1}:* Indica si el oponente del jugador del turno actual puede realizar la magia '*Envenenar'*. 0 → No; 1 → Sí.
- *opponent-magic-3 {0, 1}:* Indica si el oponente del jugador del turno actual puede realizar la magia '*Golpe ligero'*. 0 → No; 1 → Sí.
- *opponent-magic-4 {0, 1}:* Indica si el oponente del jugador del turno actual puede realizar la magia '*Golpe crítico'*. 0 → No; 1 → Sí.

Los resultados obtenidos en este último experimento para los clasificadores *JRip* y *J48* son los siguientes:

Number of Rules:	30	TOTAL	%
Correctly Classified Instances		1799	89.95 %
Incorrectly Classified Instances		201	10.05 %
Total Number of Instances		2000	

Tabla 4-15: Fase 2 - Experimento 5a: JRip – Eliminando magias oponente

Size of the tree:	207	TOTAL	%
Correctly Classified Instances		1803	90.15 %
Incorrectly Classified Instances		197	9.85 %
Total Number of Instances		2000	

Tabla 4-16: Fase 2 - Experimento 5b: J48 – Eliminando magias oponente

Estos últimos **resultados** muestran una nueva mejoría con respecto a los anteriores, ya que aunque en el número de reglas, tamaño del árbol y aciertos obtenidos para las magias, no se ha producido ningún cambio relevante, sí ha aumentado sensiblemente el índice de aciertos hasta rondar el 90%, considerándose un porcentaje suficientemente elevado como para determinar que se ha cumplido el objetivo deseado en la fase de experimentación, pasando en este punto a implementar el jugador automático en base a las clasificaciones generadas.

4.5.4 Implementación del jugador automático

Una vez finalizado el proceso de experimentación, el último paso a realizar es llevar a cabo la **implementación del jugador automático en función de las clasificaciones generadas**. Todos los experimentos del apartado anterior se han realizado tanto para el clasificador *JRip* como para el *J48*, de forma que en este momento se debe elegir uno de los resultados obtenidos que será el que se utilice para la implementación.

Finalmente se ha decidido que **el clasificador que se implementará será el de reglas de decisión *JRip***, ya que aunque el índice de aciertos es ligeramente peor que para el clasificador *J48*, los aciertos obtenidos para las magias, que es un factor clave, son muy superiores para *JRip*. Además también hay un motivo práctico para esta decisión, ya que es más sencillo de implementar, y de menor tamaño, el conjunto de reglas obtenidas que el árbol de decisión.

Para llevar a cabo la implementación de las reglas de decisión obtenidas se ha decidido crear un **método propio en la clase *PlayerIA* llamado *RunJRip()***, que recibe una gran cantidad de parámetros que definen completamente el estado del combate, y devuelve un índice que representa una determinada acción a realizar. Los parámetros que recibe son los siguientes:

- *count*: Número de intento de movimiento para el jugador automático del turno actual [0… n]. Su valor habitual debería ser 0, ya que valores superiores a 0 indican que el método ya ha sido llamado anteriormente y la acción que devolvió no era correcta. Este valor permite la posibilidad de establecer un nuevo mecanismo de selección de acción en caso de que el principal haya devuelto una jugada incorrecta por alguna causa.
- *currentTurn*: Indica el turno actual [1 ó 2]. Permite la posibilidad de establecer distintos comportamientos del jugador automático para cada uno de los dos jugadores.
- *statusPlayer[] y statusOpponent[]*: Estatus del jugador del turno actual y del oponente a este, respectivamente. Los atributos que se guardan en dicho estatus son: vida, maná, rojo, amarillo, verde y azul.
- *statePlayer[] y stateOpponent[]:* Estados en los que se encuentra el jugador del turno actual y el oponente de éste, respectivamente. Para el índice 0 guarda si el jugador tiene el bloqueo de escudo activado, y para el índice 1 guarda si el jugador está envenenado. Se indica con un 1 en caso de que esté activo dicho estado y con un 0 en caso contrario.
- *magicPlayer[] y magicOpponent[]:* Arrays que indican mediante un 1 que el jugador o el oponente respectivamente, tiene suficiente maná como para utilizar dicha magia, y con un 0 en caso de que no pudiera.
- *boxesTo1Exchange[,] y boxesTo2Exchange[,]:* Arrays que guardan si se puede obtener un determinado tipo/combinación de tipos, indicándolo con un 0 en caso negativo, con un 1 en caso de que se pueda realizar una línea de 3 casillas, y con un 2 en caso de que se pueda realizar una línea de 4 ó 5 casillas. Si dicho valor es un 1 ó 2 se guardarán también cuáles son las casillas de la fila más baja posible a intercambiar para obtener dicho tipo/combinación de tipos.
- *intBoard[,]*: Representación numérica del tablero. La correspondencia es la siguiente: ataque → 0, maná → 1, rojo → 2, amarillo → 3, verde → 4, azul → 5 y basura → 6.

El valor devuelto se corresponde con el del atributo *action1Exchange*, utilizado en el log *log1Exchange*, que representa unívocamente todas las posibles acciones que se pueden realizar en el combate para este tipo de log en concreto. Puede tomar los siguientes valores:

  - *[0 - 4]:* Indica que se ha realizado una determinada magia.
  - *[5 - 11]:* Indica el tipo de línea generada con el movimiento.

Por último, la **codificación de las reglas** en dicho método se realiza en el orden obtenido en el clasificador, comprobándolas **secuencialmente** hasta que una de ellas cumpla las condiciones establecidas y devuelva el resultado asociado a la misma.

Adicionalmente, se realiza un **control de integridad** para cada una de las reglas, con la finalidad de evitar que la acción que se devuelve no se pueda realizar en ese momento. De esta forma, si la acción asociada a dicha regla es, por ejemplo, realizar una línea roja, se comprueba que realmente existan líneas rojas en el tablero. En caso de no ser así se continúa comprobando el resto de reglas hasta llegar a alguna que cumpla las condiciones y se pueda realizar.

En el extraño caso de que después de evaluar todas las reglas no haya **ninguna que cumpla las condiciones** y se pueda realizar, se ha procedido a comprobar secuencialmente todas las acciones posibles hasta obtener alguna que se pueda realizar. Siempre debe existir alguna acción a realizar ya que, en el caso del tablero, si se detecta que no existe ningún posible movimiento, se borra y se genera uno nuevo que sí contenga algún movimiento.

Por último **tras obtener la acción a realizar**, se realiza un proceso de **perfeccionamiento** para tratar de obtener mejores resultados. Se aplica únicamente a los movimientos de casillas en el tablero y consiste en, a partir de la acción elegida, determinar cuáles serán las casillas concretas a mover para satisfacer dicha acción. Para ello se comprueba si existe algún par de casillas que al intercambiarlas generen una **combinación de tipos** que impliquen al obtenido en la acción a realizar para, en caso afirmativo, generar un movimiento que no solo obtenga dicho tipo, sino un extra con líneas de otro tipo de casillas. En todos los casos además, se intercambian las casillas de la línea más baja posible de entre todas las que cumplan el tipo elegido con el clasificador.

4.5.5 Reglas de decisión obtenidas

Las reglas de decisión que se codificarán en el jugador automático del videojuego *Magic Quest* se corresponden con las obtenidas en el *experimento 5a* de la *fase 2*, correspondiente a aplicar el clasificador *JRip* para el log *log1Exchange*. Dichas reglas son las siguientes:

```
(player-magic-4 = 1) and (opp-healt-minus-player-mana <= 1) and (opponent-
have-shield = 0) and (opponent-magic-0 = 0) => action-1Exchange=4 (21.0/3.0)

(player-healt-minus-opp-mana <= 4) and (can-get-waste = 1) and (opp-healt-
minus-player-mana >= 7) and (player-is-poisoned = 0) => action-1Exchange=1
(23.0/6.0)

(player-healt-minus-opp-mana <= 3) and (player-magic-1 = 1) and (can-get-red =
1) and (player-is-poisoned = 0) and (player-have-shield = 0) and (can-get-blue
= 1) => action-1Exchange=1 (8.0/0.0)

(player-health <= 9) and (player-magic-1 = 1) and (can-get-blue = 0) and
(player-healt-minus-opp-mana >= -3) => action-1Exchange=1 (8.0/3.0)
```

```
(player-healt-minus-opp-mana <= 1) and (player-magic-1 = 1) and (can-get-waste
= 1) and (player-magic-0 = 0) and (player-health <= 9) => action-1Exchange=1
(5.0/1.0)
(player-healt-minus-opp-mana <= 4) and (can-get-green = 1) and (player-health
>= 19) and (player-healt-minus-opp-mana <= 0) => action-1Exchange=1 (8.0/3.0)
(can-get-waste = 1) and (can-get-red = 0) and (can-get-green = 0) and (can-
get-yellow = 0) => action-1Exchange=11 (62.0/23.0)
(can-get-waste = 2) and (can-get-red = 0) and (player-healt-minus-opp-mana >=
4) and (can-get-yellow = 0) => action-1Exchange=11 (17.0/4.0)
(can-get-waste = 1) and (opponent-health <= 12) => action-1Exchange=3
(34.0/11.0)
(can-get-waste = 1) and (opponent-health <= 21) and (player-health <= 12) and
(player-magic-0 = 0) => action-1Exchange=3 (13.0/3.0)
(opponent-health <= 15) and (can-get-waste = 2) => action-1Exchange=3
(8.0/1.0)
(can-get-waste = 1) and (player-is-poisoned = 1) and (player-magic-0 = 1) =>
action-1Exchange=0 (47.0/0.0)
(player-is-poisoned = 1) and (player-magic-0 = 1) and (can-get-attack = 0) =>
action-1Exchange=0 (31.0/3.0)
(can-get-waste = 1) and (player-magic-3 = 0) => action-1Exchange=0 (2.0/0.0)
(player-magic-2 = 1) and (can-get-waste = 1) => action-1Exchange=2 (67.0/11.0)
(player-magic-2 = 1) and (can-get-yellow = 1) and (can-get-green = 1) =>
action-1Exchange=2 (12.0/0.0)
(player-magic-2 = 1) and (opponent-magic-0 = 0) and (can-get-waste = 2) =>
action-1Exchange=2 (11.0/0.0)
(player-magic-2 = 1) and (can-get-red = 1) and (can-get-blue = 1) => action-
1Exchange=2 (9.0/2.0)
(player-magic-2 = 1) and (opponent-magic-0 = 0) and (can-get-red = 2) =>
action-1Exchange=2 (4.0/0.0)
(can-get-mana = 1) and (can-get-red = 0) and (can-get-yellow = 0) => action-
1Exchange=6 (152.0/2.0)
(can-get-mana = 2) => action-1Exchange=6 (16.0/0.0)
(can-get-yellow = 1) and (can-get-red = 0) => action-1Exchange=8 (225.0/17.0)
(can-get-yellow = 2) and (can-get-red = 0) => action-1Exchange=8 (50.0/2.0)
(can-get-red = 1) and (player-healt-minus-opp-mana >= 10) and (player-magic-4
= 0) => action-1Exchange=7 (138.0/4.0)
(can-get-red = 1) and (can-get-waste = 0) => action-1Exchange=7 (94.0/11.0)
(can-get-red = 2) => action-1Exchange=7 (53.0/3.0)
(can-get-green = 1) => action-1Exchange=9 (262.0/11.0)
(can-get-green = 2) => action-1Exchange=9 (37.0/2.0)
(can-get-attack = 0) and (can-get-waste = 0) => action-1Exchange=10
(289.0/0.0)
 => action-1Exchange=5 (294.0/10.0)

Number of Rules : 30
```

5 Resultados

Los resultados mostrados en este apartado corresponden a un conjunto de pruebas que se han realizado para **comprobar el comportamiento del jugador automático** implementado tras implementar su **Inteligencia Artificial**. Para ello, se han ejecutado una serie de partidas en modo automático, en el cual el *jugador 1* ejecuta el método *Run()* sin argumentos de la clase *PlayerIA,* que devuelve la primera acción posible de entre todas las existentes. Para el *jugador 2* se utiliza el método *Run()* con argumentos, que utiliza las reglas de decisión obtenidas con el clasificador *JRip* a la hora de tomar una decisión sobre la acción a realizar.

Se han ejecutado 10 partidas con ambos jugadores, con una vida inicial de 30 puntos. Los resultados de la vida final al terminar el combate se muestran en la siguiente tabla:

N° Prueba	JUGADOR 1 – Sin IA [Vida final]	JUGADOR 2 – IA JRip [Vida final]
1	0	**24**
2	0	**30**
3	0	**15**
4	0	**30**
5	0	**19**
6	0	**24**
7	0	**21**
8	0	**17**
9	0	**18**
10	0	**26**

Tabla 5-1: Resultados IA – Jugador JRip Vs Jugador aleatorio

Como muestran los resultados, el jugador automático que utiliza las reglas de decisión obtenidas con el clasificador *JRip* no solo gana el 100% de las partidas sino que además lo hace con una notable diferencia, conservando la mitad o más de la vida en todos los casos. Esto demuestra que ha sido posible obtener un comportamiento inteligente para jugar al videojuego *Magic Quest*, por lo que se deduce que el proceso de experimentación realizado en el apartado anterior ha dado sus frutos.

6 Conclusiones

El desarrollo de este PFC me ha permitido **aplicar los conocimientos adquiridos en la carrera** con el desarrollo de un sistema que en ningún momento se ha visto en la misma: un videojuego. Este proceso ha sido **amplio y complejo**, abarcando desde la selección de la herramienta, con *XNA*; hasta el desarrollo e implementación de la IA del jugador automático.

Todo el desarrollo realizado se ha llevado a cabo a partir de la plantilla inicial que proporciona *XNA*, desde la inclusión de los distintos elementos de la pantalla hasta la elaboración de la lógica del juego. Ha sido necesario **aprender el lenguaje *C#***, sobre el cual funciona *XNA*, así como **comprender la forma en la que se debe desarrollar un juego con dicha herramienta**.

Una vez finalizado el desarrollo del videojuego también se incluyó en el ámbito de este PFC el **desarrollo de la IA del jugador automático del mismo**. Para ello se utilizó el programa ***Weka***, con el cual ya se había tenido algún contacto durante la carrera, y que ha permitido llevar a cabo una fase de experimentación en la cual se han obtenido unos resultados bastante buenos, implementando finalmente un jugador automático dotado de un comportamiento inteligente.

El **resultado final** tras el desarrollo del videojuego y de su IA ha sido **ampliamente gratificante**, proporcionando una serie de conocimientos que no se habían adquirido durante la carrera y que pueden ser de utilidad para la vida laboral en el futuro.

En lo que respecta a los **objetivos** que se habían establecido para este PFC, **se evaluará hasta que punto se ha llegado a la consecución de los mismos**. Para ello, se volverán a enumerar cada uno de ellos, adjuntando las conclusiones que se pueden obtener tras el desarrollo:

1. **El videojuego desarrollado debe tener una jugabilidad aceptable**: Aunque la jugabilidad es un concepto bastante subjetivo, se determina que *Magic Quest* ha cumplido este objetivo ya que proporciona al usuario una experiencia positiva a la hora de jugar.
2. **Se permitirá elegir entre distintos tipos de personajes**: El juego ha sido desarrollado para cumplir este objetivo, ya que permite la generación de diferentes personajes con sus propias características y magias. No obstante, debido a la experimentación que se ha llevado a cabo para generar la IA del jugador automático, ha sido necesario realizar una simplificación, estableciendo un único tipo de personaje con el que se han realizado todos los experimentos. Por otra parte, respecto a la posibilidad de que los personajes suban de nivel, mejorando sus atributos y permitiéndole aprender nuevas magias, finalmente no se ha implementado, quedando pendiente como una de las líneas futuras del juego.

3. **Existirá un mapa principal para buscar misiones y combates**: Se trata de un objetivo que no se ha cumplido debido a que finalmente el juego desarrollado se ha enfocado de una forma diferente a la inicialmente prevista. Inicialmente se esperaba que tuviera un modo historia y un planteamiento con un mayor toque de rol, con un sistema de obtención de experiencia que permitiría subir de nivel, mejorando los atributos del personaje y permitiendo aprender nuevas magias. Se trataba de un sistema demasiado extenso, por lo que finalmente se ha decidido simplificar e implementar únicamente el modo combate, estableciendo el resto de características como líneas futuras.
4. **El modo combate mezclará rol con un tablero estilo *Bejeweled***: Se trata del principal objetivo realizado ya que establece el comportamiento del juego desarrollado. Se ha realizado tal y como se definía en el objetivo, basándose en el juego de tablero de *Bejeweled* pero destinando su uso para simular un combate 2 jugadores por turnos, de modo que las líneas generadas permitan obtener maná que será necesario para utilizar las distintas magias que tiene cada uno de los personajes.
5. **Se generará un log para guardar el desarrollo de los combates**: Se trata de otro de los objetivos principales que se ha cumplido. Finalmente se han generado 3 tipos de logs que guardan, con distintos matices, el estado del combate y la acción llevada a cabo en cada turno. Estos logs permitirán obtener mediante técnicas de Inteligencia Artificial el comportamiento del jugador automático.
6. **Se desarrollará un jugador automático con técnicas de IA**: Se ha desarrollado el comportamiento del jugador automático mediante técnicas de Inteligencia Artificial. Para ello se han utilizado técnicas de aprendizaje automático con el programa *Weka* a partir de los logs generados. Por lo tanto se considera que este objetivo también se ha cumplido.
7. **Se incorporará un modo para 2 jugadores**: Se ha cumplido dicho objetivo. De hecho finalmente es el modo *2 jugadores* el único sistema que se ha desarrollado, abandonando la idea inicial de realizar un modo historia. Adicionalmente se han incluido también un modo *Jugador Vs IA* y *Automático*, que permite partidas con ambos jugadores automáticos.
8. **Debe ser todo lo sencilla posible la posibilidad de continuación del desarrollo del videojuego por terceros**: Se trata de nuevo de un objetivo bastante subjetivo pero se han realizado todos los esfuerzos posibles para conseguirlo, estructurando convenientemente el código y comentando todas las clases, métodos y algoritmos.
9. **El idioma de desarrollo del videojuego debe ser el inglés**: Se ha cumplido lo establecido en el objetivo, generando en inglés todo el código desarrollado y en español todos los comentarios.

7 Líneas futuras

Una vez expuestas las conclusiones, se pueden obtener una serie de **líneas futuras en función de los objetivos que no han sido cumplidos**. También se han recopilado otra serie de líneas futuras que no se extraen de los objetivos pero que sería interesante desarrollar. Estas son las líneas futuras de *Magic Quest*:

- **Introducir melodías y sonidos**: *Magic Quest* es un juego que se puede considerar como completo, con una jugabilidad aceptable, ya que proporciona al usuario una experiencia positiva a la hora de jugar. No obstante, no se ha introducido ningún tipo de melodía ni sonido en el juego, debido principalmente a la dificultad para obtener este tipo de elementos. Por lo tanto modificar este aspecto del juego podría proporcionar una experiencia de juego aún más enriquecedora.
- **Modificar la IA del jugador automático**: El código que establece el comportamiento de la IA del jugador automático está encapsulado el método *Run()* de la clase *PlayerIA*. Gracias a esta independencia, se podrá modificar fácilmente el código de dicho método para alterar el comportamiento de la IA, que actualmente está basado en las reglas de decisión generadas por el algoritmo *JRip*.
- **Modificar la interfaz del juego**: La interfaz gráfica está formada por determinados *sprites*, que podrán ser reemplazados por otros diferentes para modificar la interfaz del juego. Además, se podrá modificar fácilmente las dimensiones y localización en pantalla de dichos elementos ya que sus datos están centralizados en la clase estática *Config*.
- **Introducir nuevos personajes y magias**: Actualmente existen 2 tipos de personajes, aunque con las mismas características y magias, ya que simplificaba el proceso de experimentación de la IA. No obstante, el código desarrollado se ha estructurado de forma que los distintos personajes existentes, así como sus características y magias asociadas, se encuentran definidos en la clase estática *Config*, permitiendo de una forma sencilla introducir y modificar los personajes del juego.
- **Permitir elegir entre distintos tipos de personajes**: El juego ha sido desarrollado teniendo en cuenta esta posibilidad, ya que como se indica en el punto anterior, permite la generación de diferentes personajes con sus propias características y magias. No obstante, sería útil introducir una pantalla al inicio del juego en la que se permita elegir manualmente el tipo de personaje con el que se desea jugar la partida.
- **Introducir un mapa principal para buscar misiones y combates**: Aunque era uno de los objetivos iniciales, finalmente no se ha incluido un modo historia debido a que se trataba de una tarea demasiado extensa tras todo el desarrollo ya realizado. No obstante, queda como una de las principales líneas futuras, ya que aportaría un argumento y ensalzaría el toque de rol, generando un juego más adictivo y completo.

8 Gestión de proyecto

En este apartado se pretende establecer una planificación que permita llevar a cabo un control del tiempo y de los costes necesarios para el desarrollo del videojuego *Magic Quest*. En primer lugar se realizará una descomposición de las distintas fases que se abordan a lo largo del proyecto y se realizará una planificación mediante un diagrama de Gantt; posteriormente se establecerá un inventario de los costes asociados al desarrollo.

8.1 Planificación

Se han identificado una serie de fases que abarcan el completo desarrollo de Magic Quest. Se trata de las siguientes:

- **Análisis del problema**: Realizar un análisis del problema permite obtener una idea más detallada de lo que se quiere realizar.
- **Estudio inicial**: Abarca la recopilación y el estudio de la información necesaria para comprender el lenguaje y la herramienta que se utilizarán para el desarrollo.
- **Diseño de la aplicación**: Establece la arquitectura del sistema y la definición de los distintos componentes de los que se forma.
- **Implementación de la aplicación**: Fase en la que se lleva a cabo todo el desarrollo del código de la aplicación construida.
- **Desarrollo del jugador automático**: Abarca todo lo relacionado con la fase de generación e implementación de la IA del jugador automático.
- **Redacción de la memoria**: Redacción de toda la documentación necesaria para el desarrollo y la entrega del PFC.
- **Generación de la presentación**: Generación de la presentación utilizada para la lectura del PFC.

Establecidas las distintas fases de las que consta el proyecto se lleva a cabo una planificación de las mismas, con la finalidad de llevar a cabo un control del tiempo necesario para la realización del PFC. Dicha planificación se muestra en la ilustración de la siguiente página.

	Nombre de tarea	Duración	Comienzo	Fin	Pre
1	Análisis del problema	5 días	lun 01/02/10	vie 05/02/10	
2	**Estudio inicial**	**30 días**	**lun 08/02/10**	**vie 19/03/10**	**1**
3	Estudio tutoriales XNA	20 días	lun 08/02/10	vie 05/03/10	
4	Pruebas con XNA	20 días	lun 22/02/10	vie 19/03/10	
5	**Diseño de la aplicación**	**40 días**	**lun 22/03/10**	**vie 14/05/10**	**2**
6	Diseño de la arquitectura	10 días	lun 22/03/10	vie 02/04/10	
7	Definición de módulos	30 días	lun 05/04/10	vie 14/05/10	6
8	**Implementación de la aplicación**	**40 días**	**lun 17/05/10**	**vie 09/07/10**	**5**
9	Capa Controladores	10 días	lun 17/05/10	vie 28/05/10	
10	Capa nucleo	35 días	lun 17/05/10	vie 02/07/10	
11	Capa suplementos	25 días	lun 07/06/10	vie 09/07/10	
12	**Desarrollo del jugador automático**	**25 días**	**lun 12/07/10**	**vie 13/08/10**	**8**
13	Generación de los datos	10 días	lun 12/07/10	vie 23/07/10	
14	Experimentación con Weka	10 días	lun 26/07/10	vie 06/08/10	13
15	Implementación del resultado	5 días	lun 09/08/10	vie 13/08/10	14
16	Redacción de la memoria	100 días	lun 17/05/10	vie 01/10/10	
17	Generación de la presentación	10 días	lun 04/10/10	vie 15/10/10	16

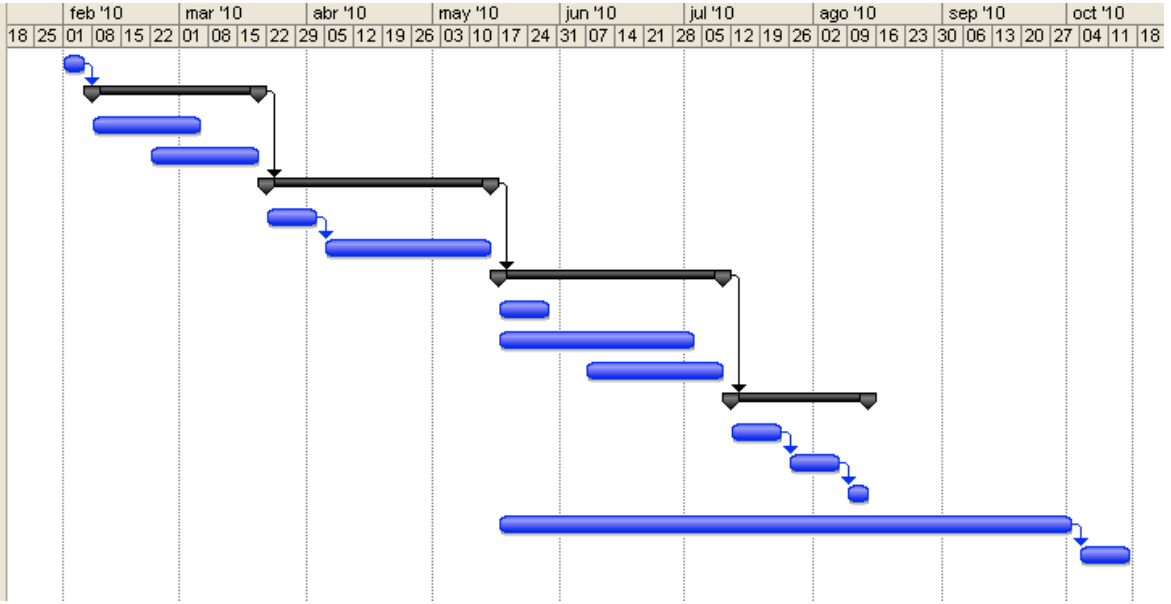

Ilustración 8-1: Planificación PFC

La ilustración anterior establece la planificación del PFC *Magic Quest*, especificando la duración y los plazos para cada una de las tareas que lo comprenden. Algunas de las tareas se detallan más específicamente al dividirla en las subtareas de las que se compone. Se da el caso de que algunas subtareas se solapan en el tiempo, dedicando durante esos días un 50% de esfuerzo para cada una de ellas. Lo mismo ocurre con la tarea de *Redacción de la memoria*, cuya planificación es de 100 días de desarrollo, debido a que gran parte del tiempo el esfuerzo dedicado a la misma será muy bajo al solaparse con otras tareas. Por último, existen dependencias entre algunas de las tareas ya que la realización de alguna de ellas no se puede comenzar sin haber finalizado por completo la tarea de la que depende.

En lo que respecta a los plazos, la duración total estimada en la planificación para este proyecto es de ocho meses y medio aproximadamente, con un ritmo de trabajo relativamente suave, correspondiente a unas 3 horas de dedicación diarias, debido a que la realización del mismo se ha llevado a cabo compaginándolo junto con una jornada laboral.

8.2 Presupuesto

En esta sección se incluyen los costes totales aplicables a la realización del PFC, desglosados de acuerdo a su tipo. Todas las cantidades expuestas en este apartado están en expresadas con la divisa euro.

8.2.1 Costes de personal

Los costes del personal se calculan aplicando una tarifa de 50 euros/hora, correspondiente a los ingenieros superiores en informática, con respecto al número de horas estimadas para la realización de cada fase del proyecto, de acuerdo a la planificación previamente establecida. Se ha considerado que cada día de la planificación se corresponde con 3 horas de trabajo.

La fase correspondiente a la redacción de la memoria, que tenía una estimación de 100 días, debe ajustarse a la hora de calcular el presupuesto ya que se solapa junto con otras fases y no se puede añadir el precio para ambas. De esta forma se estima que esta fase tiene una duración propia de 30 días.

TAREA	DURACIÓN (Horas)	Coste (€)
Análisis del problema	**15**	**750**
Estudio inicial	**90**	**4.500**
Diseño de la aplicación	**120**	**6.000**
Implementación de la aplicación	**120**	**6.000**
Desarrollo del jugador automático	**75**	**3.750**
Redacción de la memoria	**90**	**4.500**
Generación de la presentación	**10**	**500**
TOTAL	**520**	**26.000**

Tabla 8-1: Costes de personal

8.2.2 Costes de equipamiento

Los costes de equipamiento incluyen el material empleado en la realización del proyecto. El único equipo necesario ha sido el ordenador de trabajo, comprado expresamente para la realización del mismo y cuyo coste asciende a 650 €. Por otra parte hay que mencionar que no existen costes de software debido a que todas las herramientas utilizadas tanto para la fase de desarrollo como para la experimentación de la IA son gratuitas.

8.2.3 Costes totales

Establecidos los diferentes costes atribuibles al proyecto, se calcula el coste total de los mismos:

Concepto	Coste (€)
Costes de personal	**26.000**
Costes de equipamiento	**650**
TOTAL	**26.650**

Tabla 8-2: Costes totales

El coste total del proyecto asciende a **veintiséis mil seiscientos cincuenta euros (26.650 €)**.

9 Bibliografía

Las referencias utilizadas para la realización de este proyecto son las siguientes:

- http://es.wikipedia.org/wiki/Historia_de_los_videojuegos http://indicelatino.com/juegos/historia/: Información correspondiente a la historia de los videojuegos utilizada en el estado de la cuestión.
- http://www.riemers.net/eng/Tutorials/XNA/Csharp/series2d.php: Tutorial de XNA en el que explican el desarrollo de un sencillo juego 2D
- http://argade.blogspot.com/2010/05/instalacion-de-xna-y-c.html: Guía para la instalación de XNA
- http://argade.blogspot.com/2010/05/tutorial-xna-introduccion-por-argade.html: Introducción técnica de la arquitectura de XNA
- http://xna-para-torpes.blogspot.com/search/label/Teoria: Descripción de la plantilla inicial al generar un nuevo proyecto con XNA.
- http://juank.black-byte.com/tag/xna/: Código útil de C#.
- http://en.wikipedia.org/wiki/Puzzle_Quest: Wikipedia de *Puzzle Quest*.
- http://www.infinite-interactive.com/index.php: Página oficial de la compañía *Infinite Interactive*, desarrolladora de *Puzzle Quest*.
- http://www.edepot.com/forums/viewtopic.php?f=7&t=1538: Como hacer una línea de 8 casillas en *Puzzle Quest*.
- http://es.wikipedia.org/wiki/Aprendizaje_autom%C3%A1tico: Wikipedia de aprendizaje automático.
- http://the-geek.org/docs/algen/: Algoritmos genéticos y computación evolutiva.
- http://eddyalfaro.galeon.com/geneticos.html http://di002.edv.uniovi.es/~alguero/eaac/eaac_archivos/09-10/Trabajos%20para%20evaluaci%C3%B3n/Quintairos/Art%C3%ADculos%20proporcionados/temageneticos.pdf: Información detallada de las características y funcionamiento de los algoritmos genéticos.
- http://euitio178.ccu.uniovi.es/~gc2/tema_9.htm: Redes de neuronas.
- http://www.lsi.us.es/redmidas/Capitulos/LMD27.pdf: Reglas de decisión.
- http://www.cs.waikato.ac.nz/ml/weka/index_documentation.html: Documentación sobre el funcionamiento de la herramienta Weka.

Anexo I: Componentes a instalar para usar XNA y requisitos mínimos

En este anexo se realiza una descripción de los distintos componentes a instalar para poder desarrollar con *XNA* o bien para poder ejecutar las aplicaciones desarrolladas. Adicionalmente también se especifican los requisitos mínimos para que dicha herramienta funcione de manera óptima.

Los componentes necesarios para **desarrollar** con *XNA* son los siguientes:

- ***XNA Game Studio 3.0***: contiene las APIs necesarias para trabajar con *XNA*. La versión actual del componente es la 4.0, aunque para el desarrollo del proyecto se ha utilizado la 3.0 ya que se trataba de última versión en el momento de comenzar el desarrollo.
- ***Visual C# 2008 Express Edition***: es el entorno de desarrollo con el que se integra *XNA*, y permite compilar y ejecutar las aplicaciones de forma muy sencilla. Actualmente existe la versión 2010 Express, aunque en este proyecto se ha utilizado la versión 2008 Express por la misma razón que en el punto anterior.
- ***.NET Framework 3.5***: es un marco de trabajo que incluye todo lo necesario para poder correr aplicaciones basadas en *.NET*. En este proyecto se ha usado la versión 3.5, pero puede utilizarse a partir de la 2.0 en función de la versión de *XNA* instalada. Lo ideal es tener siempre las versiones más actualizadas de todos los componentes.

Los componentes necesarios para poder **ejecutar** aplicaciones desarrolladas con *XNA* son los siguientes:

- ***XNA Framework Redistributable 3.0***: provee de los componentes en tiempo de ejecución necesarios para ejecutar juegos en *Windows* que fueron desarrollados con *XNA*. La versión más actualizada en el momento de desarrollo del proyecto es la 3.0.
- ***DirectX 9.0 Runtime***: es un conjunto de tecnologías diseñado para ejecutar y mostrar aplicaciones ricas en contenidos multimedia (sonidos, imágenes, vídeo, animaciones en 3D, etc.). Contiene APIs que son usadas por *XNA* para mostrar en pantalla el contenido multimedia.
- ***.NET Framework 3.5***: es un marco de trabajo que incluye todo lo necesario para poder ejecutar aplicaciones basadas en *.NET*. En este proyecto se ha usado la versión 3.5, pero puede utilizarse a partir de la 2.0 en función de la versión de *XNA* instalada. Lo ideal es tener siempre las versiones más actualizadas de todos los componentes.

Para un funcionamiento óptimo en el desarrollo y ejecución son necesarios los siguientes **requisitos mínimos**:

- **512 MB de RAM**.
- **Tarjeta gráfica que soporte *DirectX 9.0* o superior**, compatible con shaders 1.1 como mínimo. Son compatibles las tarjetas de la serie FX de *Nvidia GeForce* y las *ATI Radeon 9500* o superiores.
- **Espacio libre en disco de al menos 2 GB**.
- **Sistema Operativo *Windows*** *XP SP3*, *Windows Vista* o *Windows 7*.
- **Otros requisitos hardware dependen totalmente del juego desarrollado**, siendo más exigentes aquellos juegos con mayores cálculos y mejores gráficos.

Puede encontrarse una guía para la instalación de *XNA Game Studio 3.1* y *Visual C# 2008 Express Edition* en el siguiente enlace:

http://argade.blogspot.com/2010/05/instalacion-de-xna-y-c.html

Anexo II: Manual de usuario

Este anexo pretende establecer una referencia a la hora de jugar al videojuego *Magic Quest*, explicando sus características y los elementos relevantes, de forma que cualquier persona adquiera el conocimiento sobre el funcionamiento y los objetivos del juego.

- **Menú inicial:**

Al comienzo de la ejecución del juego se muestra un menú inicial que consta de un conjunto de botones que permiten elegir el **modo en el que se quiere jugar la partida**. La siguiente ilustración muestra la pantalla de dicho menú:

Ilustración II-1: Menú inicial - Magic Quest

Los modos de juego de los que consta *Magic Quest* son los siguientes:

- **Modo 1 Jugador**: Establece un sistema de **juego clásico** basado en el juego *Bejeweled*, que consta únicamente de un tablero y cuyo objetivo es el de generar el mayor número de líneas posibles antes de que se agote el tiempo establecido.
- **Modo 2 Jugadores, Jugador Vs IA y Automático**: Comparten el mismo sistema de juego pero variando el tipo de los jugadores del combate. El modo *2 jugadores* establece 2 Jugadores humanos, el modo *Jugador Vs IA* un humano y el otro controlado por la IA, y el *Automático* 2 controlados por la IA. El sistema de juego consiste en destinar el uso del tablero *Bejeweled* para simular un **combate 2 jugadores** por turnos cuyo objetivo es reducir la vida del jugador contrario a 0.

- **Modo 1 Jugador:**

Se trata de un modo de juego clásico basado en el juego *Bejeweled*, que consta únicamente de un tablero y cuyo objetivo es el de **generar el mayor número de líneas posibles** antes de que se agote el tiempo establecido. La siguiente ilustración se corresponde con una pantalla de juego en dicho modo:

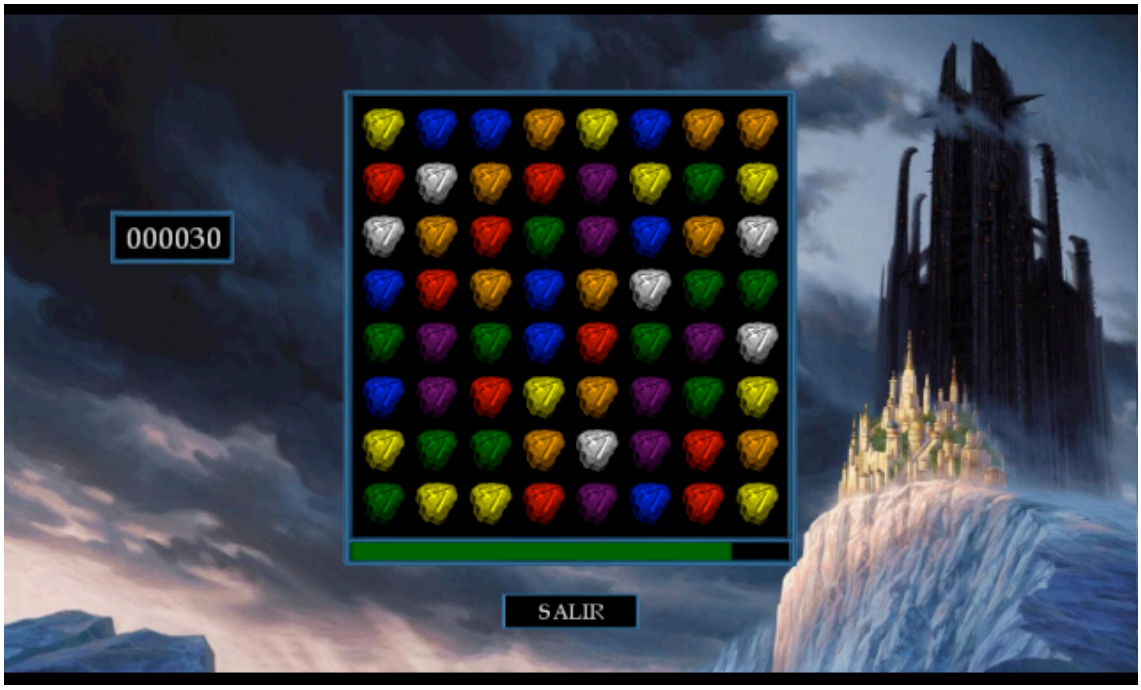

Ilustración II-2: Modo 1 Jugador – Magic Quest

Como se puede observar, el elemento principal de este modo es el **tablero**, que establece un conjunto de casillas de diferentes colores distribuidas al azar. El **cometido del jugador** es intercambiar dos casillas que se encuentren situadas una al lado o encima de la otra, de forma que la nueva distribución de las casillas **produzca una línea de 3 o más casillas** del mismo color. Por ejemplo, en la ilustración anterior, se podrían intercambiar la casilla situada en la esquina inferior izquierda con la que tiene situada encima para generar tanto una línea de color amarillo como otra de color verde.

Las casillas correspondientes a las líneas generadas **se eliminan del tablero y se desplazan hacia abajo** todas las situadas en la parte superior a las mismas, de forma que se podrían generar nuevas líneas en cascada, continuando con el proceso hasta que ya no se genere ninguna. **Cada casilla eliminada** suma **1 punto** en el marcador y produce un **leve incremento de la barra de tiempo**, que es la barra verde situada justo debajo del tablero.

La barra de tiempo se reduce paulatinamente de forma que el objetivo del jugador sea **sumar la mayor cantidad de puntos posibles** antes de que se reduzca por completo la barra de tiempo.

- **Modo 2 Jugadores, Jugador Vs IA y Automático:**

Se trata de modo que comparten el mismo sistema de juego pero variando el tipo de los jugadores del combate de la siguiente forma:

- ***Modo 2 Jugadores:*** El control de ambos jugadores se deja en manos de personas. Un marco dorado alrededor de la imagen del personaje indica a quien le corresponde el control en ese determinado turno.
- ***Modo Jugador Vs IA:*** Se establece el control del jugador 1 a una persona y del jugador 2 a un jugador automático controlado por la IA. El turno correspondiente a la persona se representa con un marco dorado alrededor de la imagen de su personaje, y el correspondiente al jugador automático con un marco rojo.
- ***Modo Automático:*** El control de ambos jugadores se deja en mano de la IA del juego. Un marco rojo alrededor de la imagen del personaje indica a quién le corresponde el control en ese determinado turno.

El sistema de juego consiste en destinar el uso del tablero *Bejeweled* para simular un **combate 2 jugadores** por turnos cuyo objetivo es **reducir la vida del jugador contrario a 0**, bien **realizando líneas** que produzcan al enemigo un daño directo, o bien **utilizando alguna magia** que penalice al enemigo o reduzca su vida. Para poder utilizar las diferentes magias será necesario obtener sus correspondientes cantidades de los distintos tipos de maná requeridas, y éstas se consiguen realizando líneas de casillas de diferentes tipos. La siguiente ilustración se corresponde con una pantalla de juego en dicho modo:

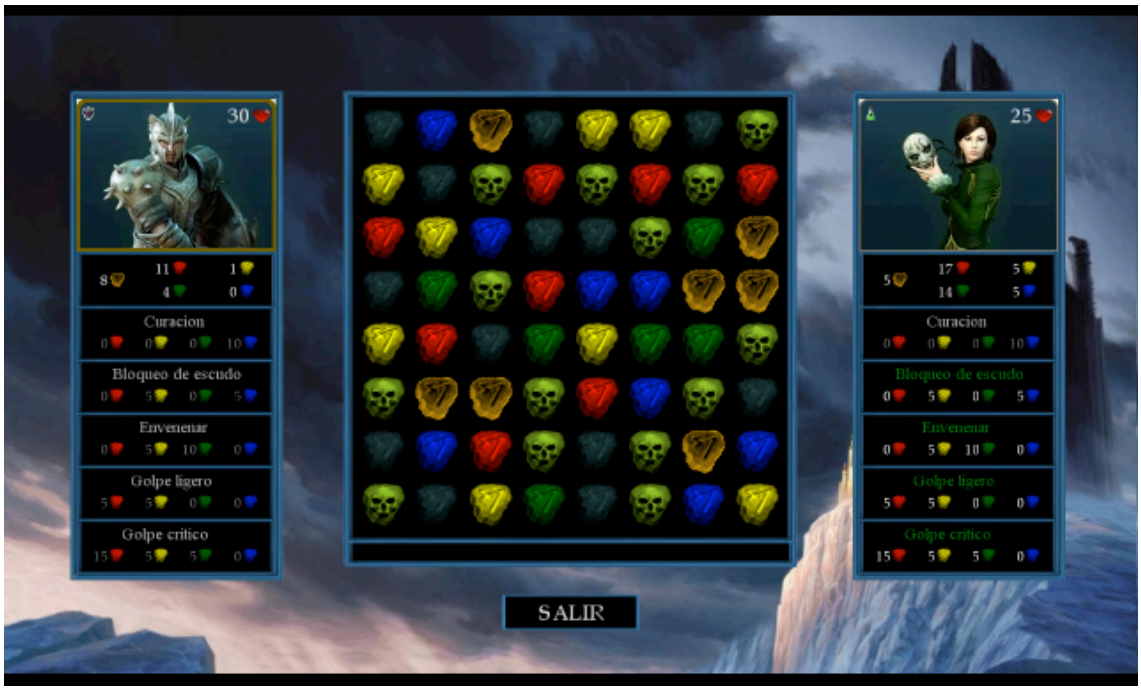

Ilustración II-3: Modo Vs – Magic Quest

Este sistema de juego tiene de nuevo como uno de los elementos relevantes el **tablero**, pero también contiene el estatus y las magias de los **2 jugadores** entre los que se desarrolla el combate.

El **tablero** genera diferentes **tipos de casillas** en este caso que para el modo *1 Jugador*, pudiendo agruparse en los siguientes tipos:

- **Maná** (rojo, amarillo, verde, azul): Permiten al jugador acumular maná del correspondiente color de la casilla, que será necesario para poder utilizar las magias del personaje.
- **Maná dorado**: Permite al jugador acumular maná de dicho tipo, el cual es utilizado para realizar la magia especial '*Golpe crítico'*. Se acumula realizando líneas de dicho tipo de maná, y también se produce un incremento adicional de 1 al realizar cualquier tipo de línea de 4 casillas, y de 2 con cualquier tipo de línea de 5 casillas.
- **Calaveras**: Producen daño directo al otro jugador al producir una línea, reduciendo su vida en 1 por cada calavera alineada.
- **Basura**: Se corresponden con las casillas con el icono difuminado de color oscuro y no tienen ninguna utilidad, ya que no permiten acumular ningún tipo de maná ni producen daño al oponente.

Los jugadores están representados por los siguientes elementos:

- **Status**: Atributos que definen las características del jugador en cada momento de la partida. Se guarda y representa la vida del jugador, así como el maná dorado acumulado, y el maná de cada uno de los colores. El status inicial de ambos jugadores es 30 de vida y 5 de cada maná.
- **Estado**: Indica si el jugador se encuentra alterado por algún estado especial. Se representa con un icono en la parte superior izquierda de la imagen del personaje, pudiendo tratarse de los siguientes:
    - *Bloqueo de escudo*: Se representa con un escudo e indica que no sufrirá daño la próxima vez que el oponente le proporcione un golpe, bien sea por alinear calaveras o por alguna de las magias.
    - *Envenenado*: Se representa con un frasco verde e indica que ese jugador está envenenado, reduciendo 2 su vida cada vez que sea su turno, hasta que el jugador utilice la magia de curación.
- **Magias**: Cada jugador posee un conjunto de magias, que podrá utilizar en su turno en lugar de realizar un movimiento en el tablero, siempre y cuando tenga acumulado el maná suficiente para realizarla, el cual se indica en la propia magia. Siempre que una magia se puede realizar aparecerá resaltada con la letra en verde y aparecerá un marco dorado al pasar por encima de ella. Adicionalmente al pasar por encima de todas ellas también se muestra una descripción de las distintas magias en el marco de debajo del tablero.

Actualmente existen los siguientes tipos de magias:

- *Curación [10 azul]*: Elimina todos los estados negativos, que actualmente sólo puede ser envenenado, y suma 5 a la vida.
- *Bloqueo de escudo [5 amarillo, 5 azul]*: Activa el estado de '*Bloqueo de escudo*' anteriormente descrito, previniendo el siguiente golpe que se fuera a sufrir.
- *Envenenar [5 amarillo, 10 verde]*: Activa en el enemigo el estado de '*Envenenado*' anteriormente descrito, reduciendo cada uno de sus turnos su vida en 2.
- *Golpe ligero [5 rojo, 5 amarillo]*: Golpe leve que reduce 6 puntos la vida del oponente.
- *Golpe crítico [x dorado, 15 rojo, 5 amarillo, 5 verde]*: Golpe fuerte que reduce la vida del enemigo el valor de maná dorado acumulado.