UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
ESCUELA DE INGENIERÍA INFORMÁTICA

Beat Fighters!

Exploración y desarrollo de diferentes técnicas de desarrollo de videojuegos multijugador

David Montesdeoca Suárez

Tutor: Agustín Trujillo Pino
Tutor: Antonio José Sánchez López

Las Palmas de Gran Canaria, Julio de 2014

Agradecimientos

A mis tutores, Agustín Trujillo y Antonio José Sánchez. Ellos han representado la luz que me ha guiado durante todo el proceso que ha sido este proyecto. A Agustín agradecerle especialmente el optimismo y las facilidades que siempre me ha aportado, desde mi primera etapa en la ULPGC hasta la actualidad. A Antonio la tranquilidad que transmite, sus grandísimas ideas (aunque no haya sabido aprovecharlas todas) y su pasión por el mundo del videojuego. Me ha permitido dar un pasito más hacía el que deseo que sea mi futuro, y atisbarlo levemente.

A mis compañeros del Máster en Diseño y Desarrollo de Videojuegos de la UCM. La vida sin ellos no sería igual. Siempre están ahí cuando se les necesita, siempre dispuestos a aportar su granito de arena, sus ideas y conocimientos.

Finalmente, a toda mi familia, por todo el apoyo y ánimo que me han dado durante el desarrollo de este trabajo, así como en todos los momentos de mi vida. En especial a mis padres, por darme la posibilidad de luchar por mis sueños, apoyarme con todo su corazón y esfuerzo en mi lucha, y saber entenderme y aguantarme en los malos momentos.

"Para obrar, el que da debe olvidar pronto,
y el que recibe, nunca"
Séneca

Índice General

Capítulo 1: Introducción

"El hombre sólo juega cuando es libre en el pleno sentido de la palabra,
y sólo es plenamente hombre cuando juega".
Friedrich Schiller ,Cartas sobre la educación estética del hombre, 1795

1.1. Motivación y Objetivos

En la actualidad, no cabe duda de que la industria del videojuego se ha ganado un puesto más que imperante en el mercado. Además, se trata de una industria que ha evolucionado vertiginosamente, desde los sencillos juegos de la era de los 8-bit hasta llegar a los juegos multijugador masivos actuales. Así como los juegos han cambiado, las técnicas de desarrollo también lo han hecho. El presente proyecto, titulado ***"Beat Fighters!: Exploración y Desarrollo de diferentes Técnicas de Desarrollo de Videojuegos Multijugador"*** es, en realidad, un estudio y desarrollo de distintas técnicas utilizadas en el desarrollo de videojuegos, haciendo hincapié en dos campos en concreto: la inteligencia artificial y la red.

1.1.1. Objetivos

En este proyecto se pretende cubrir varios objetivos. Estos objetivos son los siguientes:

1. Implementación del prototipo de un videojuego orientado a multijugador.
2. Estudio y desarrollo de distintas técnicas de inteligencia artificial orientada a videojuegos.
3. Estudio y desarrollo de distintas técnicas de desarrollo de videojuegos en red.
4. Realización de una memoria que sirva a su vez de documento orientativo para aquellas personas que deseen iniciarse en el desarrollo de videojuegos, especialmente en las áreas de inteligencia artificial y redes.

Para la consecución de este último objetivo, los capítulos dedicados a la inteligencia artificial y redes se compondrán de un apartado de análisis de técnicas y tecnologías de uso común en el desarrollo de videojuegos, seguida del apartado en el que se explican los detalles de implementación que se han seguido en el desarrollo del juego. Además, parte de la implementación de la interfaz estará orientada a dar *feedback* de lo que está sucediendo en los entresijos del código.

1.1.2. Motivaciones

En gran parte, la motivación principal para haber decidido desarrollar un videojuego en este proyecto viene de haber sido consumidor de videojuegos desde la más tierna infancia. Esta motivación fue la que me llevó a tomar la decisión de querer dedicar mi vida profesional al desarrollo de videojuegos.

Llevado por esta motivación, cursé durante el curso 2012/2013 el Máster en Diseño y Desarrollo de Videojuegos de la Universidad Complutense de Madrid, que acabé con una Mención de Honor. El proyecto realizado durante el desarrollo del máster, aunque complejo, no me permitió cubrir todas las áreas del desarrollo de videojuegos, que es ya de por sí, un reto interesante para cualquier desarrollador de software, por lo que quise aprovechar esta oportunidad para investigar y desarrollar los campos que no tuve oportunidad en el pasado y así completar mi formación.

Además, se une otro de los deseos por el que he decidido desarrollar videojuegos. En la sociedad actual, la mayoría de niños pasan horas y horas delante de la pantalla del ordenador, consumiendo videojuegos. Esta actividad no es intrínsecamente mala, siempre que

el consumo se haga de forma responsable, y los padres sean conscientes de qué tipo de videojuego consumen sus hijos y durante cuánto tiempo. Las personas aprendemos a través del juego, es algo que está en nuestra naturaleza, así que, en los videojuegos, se encuentra una gran oportunidad para educar a las generaciones futuras de forma saludable. Es por esta razón, por la necesidad de ayudar en la educación de las personas, que he decidido orientar este documento para que aquellas personas que quieran obtener unos conocimientos básicos sobre desarrollo de videojuegos, encuentren aquí un punto de partida.

En resumen, considero que el proyecto que he elegido aúna un desarrollo complejo y variado con una vocación social, lo que lo convierte en un proyecto bastante interesante. Y, por si esto fuera poco, es un juego, así que… ¡será divertido!

1.2 Introducción a la industria del Videojuego

1.2.1 Historia de los videojuegos

Todos los seres vivos aprenden las competencias básicas para su supervivencia a partir del juego. Por ejemplo, los gatos durante su infancia aprenden a cazar cazando a sus hermanos y aprenden a controlar la fuerza de sus mandíbulas mordiendo, y sobretodo recibiendo mordidas, de sus hermanos. Pero ellos simplemente están jugando. El juego es algo innato que estimula la creatividad, el pensamiento lógico, reduce el estrés y nos hace sentir más alegres. Es quizás por ser algo intrínseco a la vida y al ser humano que, según se iban realizando avances en la tecnología, un grupo de personas empezó a pensar en aplicaciones lúdicas de la misma.

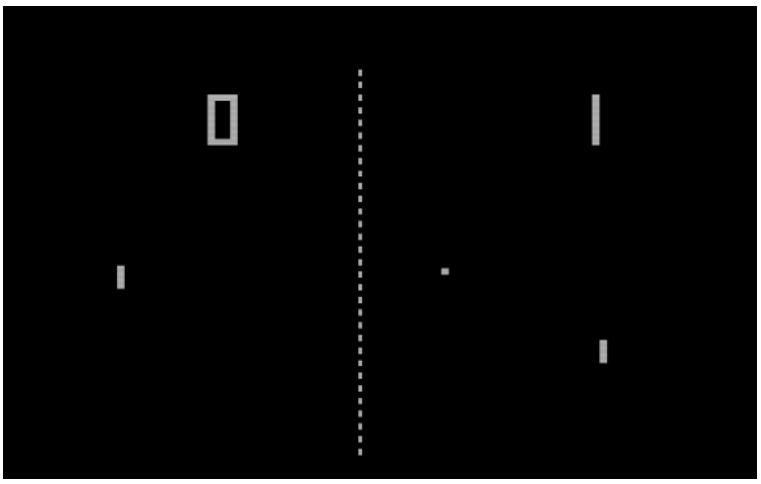
La historia de los videojuegos está dividida en generaciones, correspondiente cada cual a un cambio de la tecnología. A continuación, realizaremos un breve recorrido por la historia de los videojuegos, basándonos en el realizado en *HighScore!* [1].

En 1958, el físico William Higinbotham desarrolló *"Tennis for Two"*, que aunque no era aún un juego digital, incluía una interfaz gráfica con movimiento. Se jugaba en el denominado "*Dispositivo de Entretenimiento de Tubos de Rayos Catódicos"*[1], que en realidad era un osciloscopio modificado. Este dispositivo permitía, por vez primera, que dos personas jugaran una partida de algo parecido al tenis sin necesidad siquiera de saber que es una raqueta.

Tennis For Two

[1] www.jmargolin.com/patents/2455992.pdf

Pero no sería hasta 1972 que los videojuegos empezaran a popularizarse. El interés por los videojuegos llegó de mano del juego "*Pong"* de Atari[2]. La popularidad del juego provocó la aparición de múltiples clones por parte de distintas compañías, como Magnavox o Sears, surgiendo así la industria del videojuego. Pero llegó hasta tal punto el entusiasmo por crear clones de "*Pong"* que el mercado acabó por saturarse, provocando la desaparición de la mayoría de compañías que intentaron subirse al tren. Este evento marca la **primera crisis de los videojuegos**, datada en 1977.



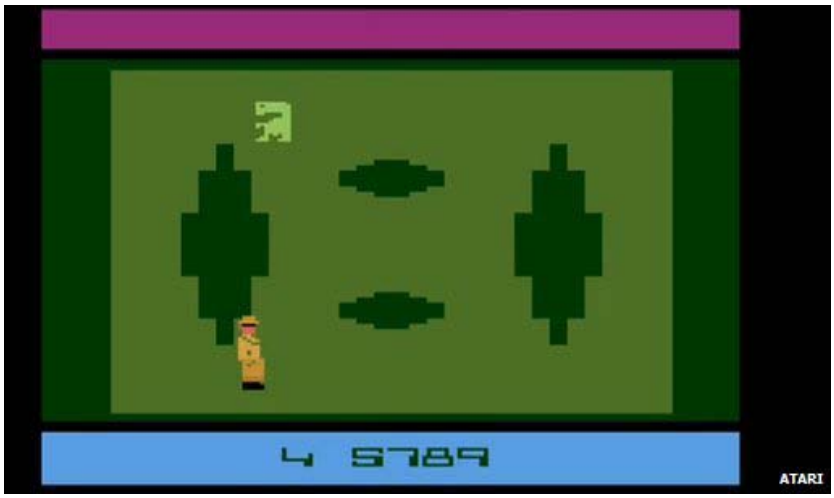
Pong

La segunda generación de consolas llegaría, aun así, en 1978, gracias en parte a la aparición de la tecnología MOS, que permitió el abaratamiento de los costes de producción. La *Atari 2600* se convirtió en la dominadora clara del mercado, siendo una de las consolas más vendidas de la historia.

Sin embargo, parte de los empleados de Atari estaban descontentos. Eran conscientes de la popularidad que estaba ganando la empresa y de los beneficios que estaba obteniendo a costa de su trabajo, mientras veían que sus sueldos permanecían congelados. Esto llevó a que parte de estos empleados abandonaran Atari en 1979 y fundaran Activision[3], la primera desarrolladora third-party (es decir, empresa que desarrolla juegos para cualquier sistema sin mantener exclusividad con ninguna de ellas). Atari viviría su episodio más amargo en torno a 1983, en la conocida como **segunda crisis del videojuego** o **debacle de Atari.**

Diversos son los factores que provocaron esta segunda crisis. Por una parte, en el mercado existía un exceso de consolas, cada uno con su propio y amplio catálogo de juegos, que provocaba en los clientes confusión al no saber cuál debían comprar. Por otra parte, Atari invierte importantes sumas en marketing para sus dos próximos grandes lanzamientos: una versión casera de *"Pac-Man"* y la adaptación de la popular película *"E.T. El Extraterrestre"*, que acaban convirtiéndose en sendos fracasos pues, con la intención de ser vendidos en campañas navideñas, se sacó al mercado dos productos inacabados y llenos de bugs, que resultaron ser sendos fracasos de ventas. Incluso existe una leyenda urbana dentro de la industria según la cual los ejemplares no vendidos de "*E.T."* se encuentran enterrados en algún lugar del desierto de Texas.

[2] http://www.atari.com/es
[3] http://www.activision.com/es/



El Infame E.T. El Extraterrestre

Finalmente, una demanda interpuesta por Atari contra Activision falla a favor de esta última, dejando a Atari en un estado muy delicado, con más de 300 millones de dólares en deuda, lo que casi supuso la bancarrota de la empresa.

Es en esta época, con la aparición de la *Commodore 64* y el *ZX Spectrum*, que los ordenadores empezaron a hacer frente a las consolas. En la década de los ochenta la industria del videojuego para ordenador vive un importante auge, con la fundación de compañías emblemáticas como Electronic Arts. Al principio los juegos de ordenador eran clones de los juegos de consolas y recreativas, pero los bajos costes de desarrollo permitieron la aparición de novedosos juegos. Esto, unido a la crisis del 83, permitió a los ordenadores superar en ventas a las consolas en 1984.

Con la **tercera generación de consolas** (1985) la industria pudo regenerarse, de la mano de una nueva competidora: Nintendo[4] lanza al mercado su *Nintendo Entertainment System (NES) de 8 bits*, que junto a un espectacular catálogo de juegos, entre los que destacan clásicos de los que hoy día siguen saliendo secuelas como *"Super Mario"*, "*The Legend of Zelda"* o *"Metal Gear"*, dominó por completo el mercado, sin que la nueva *Master System* de Sega pudiera si quiera competir con dignidad.

El clásico Super Mario Bros (NES)

[4] http://www.nintendo.es/

Para competir con la *NES*, Sega[5] lanza al mercado en 1988 su consola *Sega Génesis (Mega Drive* en Europa) de 16 bits, dando el pistoletazo de salida a la **cuarta generación de videojuegos.** En respuesta, Nintendo lanza su *Super Nintendo Entertainment System* (*SNES)* con la que consigue derrotar a Sega en territorio nipón y americano, pero no en Europa. Sega realiza los primeros experimentos con un soporte que no fuera los clásicos cartuchos, lanzando la extensión *Mega CD* para su consola, mientras que Nintendo realiza el salto a las consolas portátiles con la *Game Boy*, que dominaría el mercado portátil durante dos generaciones completas.

Entre 1993 y 1999 asistimos a la denominada **quinta generación de consolas** con representación de 32 y 64 bits. Las dos grandes competidoras de esta generación serían Nintendo con su *Nintendo 64* y Sony[6] con su novedosa *PlayStation*, primera consola en apostar firmemente por el CD. Esta generación supone el salto del 2D al 3D. Por su parte, Nintendo continuó apostando por *GameBoy* lanzando versiones actualizadas como la *Game Boy Color*.

Metal Gear Solid (PSX)

Ya en la **sexta generación,** que se corresponde con los años entre 1998 y 2005, Sony consigue hacerse con el dominio absoluto del mundo de las consolas. Su *PlayStation 2* es aun a día de hoy la consola más vendida de la historia. Esto es a la vez razón y consecuencia de ser considerada la consola con el mejor catálogo de juegos de la historia. Además, no era solamente una consola de videojuegos, sino también un reproductor de DVD. Por su parte Nintendo abandonó el formato cartucho con su *Nintendo GameCube* mientras Sega afrontó su final como desarrolladora de consolas con la *Sega DreamCast*.

Entre las nuevas incorporaciones se encuentra Microsoft[7], que decide entrar en el mundo de las consolas con *XBOX,* orientada a la experiencia multijugador y con la novedad de incluir un disco duro integrado. Nokia intentó arrebatar el trono portátil a Nintendo y su nueva *GameBoy Advance,* pero su *N-Gage* no resultó ser demasiado exitosa, dando inicio y fin a la efímera trayectoria de Nokia en el mundo del videojuego.

[5] http://www.sega.es/
[6] www.sony.es/
[7] http://www.xbox.com/

Gran Turismo 4 (PS2)

Para la **séptima generación (2004-2012)** sólo tres empresas compiten por el trono a nivel mundial. Microsoft intenta adelantarse a sus competidoras con el lanzamiento en 2005, casi un año antes que sus competidoras, de *XBOX 360*. Por su parte, Sony continua fiel a su exitosa marca *PlayStation* lanzando su tercera versión. Nintendo trae la revolución con su consola *Wii,* caracterizada por un nuevo tipo de control, en el que en lugar de aporrear botones a la manera clásica, el mando estaba equipado con una serie de sensores de movimiento. Esta tecnología, junto con un catálogo de juegos orientados a la inclusión de todos los miembros de la familia, catapultó a la consola de Nintendo hacia el estrellato, obligando a Sony y Microsoft, con consolas más clásicas, a reinventar sus consolas en mitad de la generación con el fin de poder competir.

Sony sacó al mercado su nuevo control *PlayStation Move*, de características muy similares al mando de Nintendo, mientras que Microsoft decide dar un paso más y en lugar de un nuevo mando, lanza una nueva cámara, llamada *Kinect*, con tecnología de captura de movimiento, temperatura, etc.

Wii Sports (Wii)

Esta generación trae consigo otro nuevo cambio en la industria. Aparecen las plataformas de distribución online. Hasta entonces, para poder vender un juego, había que editarlo en formato físico y contratar una distribuidora que hiciese llegar a la mayor cantidad de territorios posibles. Este hecho, aparte de privar a occidente de geniales obras que sólo fueron lanzadas en territorio nipón, aumentaba los costes de producción de un videojuego. En

esta generación, aparecen tanto *PlayStation Network* como *XboxLive*, sirviendo de plataforma de distribución online. Por su parte, Valve[8] (responsable de grandes juegos como *Half Life)* lanza *Steam* como plataforma de distribución a través de internet. Este hecho favorece el nacimiento de pequeños estudios Indie que de otra forma no hubieran podido lanzar sus juegos, desarrollando auténticas joyas como *FEZ*[9].

Una joya Indie: FEZ

Por otra parte, el mercado portátil recibe, por parte de Nintendo la *NintendoDS*, una consola con dos pantallas, de las cuales una es táctil. Por parte de Sony, aparece *PlayStation Portable* (*PSP).* Los smartphones empiezan a ganar popularidad, especialmente iPhone, que con sus pantallas táctiles, acelerómetros y giroscopios hacen las delicias de los desarrolladores, que además, pueden disfrutar también de sus plataformas de distribución *GooglePlay* para Android y *AppleStore* en iPhone. Pioneros en este campo fueron Rovio, con su juego *"Angry Birds"*.

Actualmente nos encontramos en la **octava generación de videoconsolas**. Esta generación empieza en 2011 con la llegada de la portátil de Nintendo con tecnología 3D: la nueva *Nintendo 3DS.* Esta consola permite generar imágenes estereoscópicas que gracias a la tecnología *Paralax Barrier*[10] pueden ser visualizadas sin ningún tipo de dispositivo adicional. Por su parte, Sony y Microsoft han lanzado recientemente las nuevas versiones de sus consolas, *PlayStation 4* y *XBOX One* respectivamente, mientras que Nintendo ha apostado por *Wii U*, que aboga por la interoperabilidad con *Wii*.

Castlevania Mirror of Fate (N3DS)

[8] http://www.valvesoftware.com/
[9] http://fezgame.com/
[10] 3www1.cs.uic.edu\images\TomPeterka_thesis.pdf

Finalmente, la popularidad del sistema operativo *Android* está provocando la aparición de nuevas consolas, como son *OUYA y GameStick,* con este sistema como principal valedor. Está por ver hasta donde conseguirán llegar, pero sin duda, representan el futuro de la industria.

1.2.3 Desarrollo de un videojuego: Etapas y Perfiles

El proceso de desarrollo de un videojuego no difiere en demasía del desarrollo de un producto software cualquiera, aunque existen algunos matices.

Por supuesto, todo empieza por una **idea**. En ocasiones se desea hacer un clon de otro juego con algunos matices, quizás una nueva ambientación o una nueva mecánica. Otras veces se quiere hacer un producto totalmente nuevo. Durante su conferencia en el Ideame'13, Francisco Téllez, diseñador, programador y artista de "*UnEpic*" confesó, en tono de humor, que las mejores ideas surgen siempre alrededor de la cerveza, y sobre todo, que las mejoran los comentarios casi azarosos de las personas cercanas. De su intervención debe extraerse que las ideas deben ser compartidas, pues cada pequeño detalle puede suponer un gran cambio en el diseño. En concreto, comentaba Téllez que en principio, *UnEpic* iba a ser un juego serio, hasta que su mujer le sugirió darle un toque humorístico. Este detalle es el que ha convertido a *UnEpic* en un juego muy popular, y el primer juego español en salir en *Wii U*.

Una vez tenemos una idea más o menos definida, empieza la fase de **diseño.** Aquí es donde se encuentra la mayor diferencia con otros desarrollos del software. Así como en todos existe una etapa de diseño, en la que se cubren las funcionalidades que va a tener el software, en videojuegos esta etapa comprende el diseño de personajes, gameplay, mecánicas, historia… Esta etapa es crítica e iterativa, continuando aun durante las etapas de desarrollo y testeo, incluso más allá de la publicación del juego, durante la etapa de mantenimiento. Durante la primera etapa se define la mayor parte de contenido que tendrá el juego, mientras que según avance el desarrollo, se encargará de ajustar y equilibrar las distintas mecánicas para conseguir un resultado lo más pulido posible. Para más información sobre esta etapa, recomiendo la lectura de *The Art of Game Design*[2].

A continuación llega la etapa de **desarrollo**. Por supuesto, cada estudio y empresa tiene sus propios métodos de desarrollo, aunque es bastante común, sobre todo en los estudios indie, el uso de métodos ágiles, buscando tener lo antes posible un prototipo jugable que hacer crecer poco a poco. Durante esta etapa trabajarán en paralelo tanto los programadores de gameplay, los programadores de herramientas, los artistas y otros creadores de contenido. Eventualmente, durante la etapa de desarrollo se producirá un "*Vertical Slice*" del juego, que consiste en una sección completamente implementada, tal y como será en el juego final. Este *vertical slice* se suele utilizar en presentaciones en busca de financiación.

La etapa de desarrollo normalmente se fusiona con la etapa de **testeo**. Normalmente se pasa por dos etapas de testeo diferenciadas, una alpha, de testeo por parte del propio equipo del estudio, y una segunda, beta, para ser testeado por una parte de la comunidad de jugadores.

Cuando se considera que el juego está suficientemente pulido, o cuando se llega a la fecha límite contractual, se procede a la **publicación** del juego. Este es un proceso muy diferente según la plataforma destino o el método de publicación que será usado. Tras este punto, los usuarios pueden, al fin, hincarle el diente al juego y disfrutar del trabajo realizado por el equipo durante meses. De las opiniones de los jugadores el equipo recibirá el *feedback* necesario para realizar un correcto **mantenimiento** del producto, ya sea corrigiendo bugs, equilibrando mecánicas, o incluso, alargando la vida del juego añadiendo más contenido en forma de actualizaciones.

Blind Bongo[11], mi primer juego, tras el proceso de publicación.

Como se puede extraer del proceso descrito, en el desarrollo de videojuegos hay tres roles claramente diferenciados (sin contar a los jugadores):

- **Diseñador**: Se encarga del diseño de las mecánicas de juego, niveles… Todo lo que se va a ser jugable. Los buenos diseñadores están altamente cotizados, y normalmente lideran los estudios. Entre los más destacados de la industria se puede encontrar a personas como *Hideo Kojima* o *Peter Molyneux.* En la inspiración de un diseñador puede estar la diferencia entre un buen juego y una obra maestra.
- **Desarrollador**: Programadores, ingenieros de software, desarrolladores de herramientas... En general, la gente técnica. Son los encargados de hacer realidad todo lo que los diseñadores han descrito en el diseño, y de unirlo con el trabajo realizado por los artistas.
- **Artistas**: Principalmente músicos y artistas gráficos. Son los encargados de crear el contenido que dará vida al trabajo tanto de los diseñadores como de los desarrolladores. Curioso es el caso de "*Darkseed"*, un juego que es un homenaje directo a la obra de H.R.Giger, y que contó con el propio Giger para realizar gran parte del trabajo artístico del juego.

1.2.2 Motores

Antes de la existencia de los motores de videojuegos, los juegos se desarrollaban como entidades singulares. Desarrollar un juego para una consola específica implicaba realizar un diseño totalmente orientado a la consola, de forma que se pudiera hacer un uso óptimo del *hardware* de la misma. Incluso era difícil reutilizar el código entre diferentes juegos. Los motores aportan un nivel de abstracción por medio del *framework* que incluyen, en el que se proporciona a los desarrolladores las operaciones básicas para la programación de videojuegos. Entre ellas, podemos encontrar:

[11] https://play.google.com/store/apps/details?id=com.PerenquenStudio.BlindBongo

- **Motor gráfico para la renderización de gráficos 2D/3D:** Una API para la generación de gráficos en pantalla. Generalmente actúa como interfaz que facilita el uso de librerías como *OpenGL* o *Direct3D.*
- **Motor Físico:** Facilita una simulación aproximada en tiempo real de ciertos sistemas físicos, como dinámica de cuerpos rígidos o fluidos. En ocasiones es propio del motor, y en otras actúa también como interfaz para otros motores como *PhysX.*
- **Soporte de Sonido**: Una API que permite la reproducción de diferentes formatos de sonido y el ajuste de parámetros como el tono o el volumen.
- **Scripting:** Soporte para definir el comportamiento de los diferentes elementos del juego a través de un lenguaje de programación.
- **Animaciones:** Puede ser tanto la capacidad para reproducir animaciones incluidas en los propios modelos, como un sistema para la animación tradicional por frames.
- **Networking:** Conjunto de algoritmos y componentes destinados a la resolución de los problemas más comunes en la conectividad de dos equipos informáticos.
- **Gestión de Memoria e Hilos:** Herramientas de alto nivel que facilitan la gestión y control de los mismos.
- **Grafos de escena:** Ofrece un mundo virtual vacío sobre un espacio cartesiano en el que el desarrollador puede insertar modelos.
- **Multiplataforma:** Ofrecen servicios para crear ejecutables específicos que funcionen en distintas plataformas.

El objetivo de estos motores es el de simplificar el desarrollo de los videojuegos, evitando que los desarrolladores tengan que reprogramar todos los elementos que necesitan para la creación de un juego. El objetivo es, básicamente, que no tengan que reinventar la rueda.

En la actualidad, y gracias a la madurez de la tecnología de motores, no son solo usados para el desarrollo de videojuegos, sino también para crear simuladores, que pueden ser usados, por ejemplo, para el entrenamiento médico.

A continuación se resumen las características de algunos motores que existen actualmente en el mercado.

1.2.2.1 OGRE[12]

En realidad, Ogre es un motor gráfico gratuito orientado a escenas y a tiempo real, bajo licencia MIT. Su diseño lo convierte en un motor muy modular, y el que hecho de que esté

[12] http://www.ogre3d.org/

publicado bajo código abierto ha hecho que programadores externos desarrollaran utilidades suficientes para convertirlo en un motor de videojuegos. Un equipo de desarrollo oficial se centra en el motor gráfico. Actualmente puede generar ejecutables para *Linux, OS X, Windows, iOS* y *Android.*

1.2.2.2 Unreal Development Kit (UDK)[13]

Motor de videojuegos desarrollado por *Epic Games* para el juego *Unreal* publicado en 1998. Está desarrollado en C++. Su portabilidad y potencia lo han convertido en uno de los motores más utilizados. A fecha de 25 de junio de 2014 una licencia UDK cuesta 19€ mensuales, más un 5% de los beneficios del juego, permitiendo publicar tanto para *Windows* como para *Android* y *iOS.*

1.2.2.3 Torque[14]

Motor de videojuegos desarrollado inicialmente por *Dynamix* para el juego *"Tribes 2"*. Posteriormente pasó a llamarse *Torque 3D* y pasó a ser desarrollado por la compañía *GarageGames.* Al igual que Ogre, se liberó bajo licencia MIT en 2012, por lo que también es gratuito. La versión 3D no soporta ninguna plataforma móvil.

1.2.2.4 CryEngine[15]

Motor desarrollado por la empresa *Crytek,* originalmente un motor de demostración para la empresa *Nvidia*, que, debido a su potencial, se utilizó por primera vez en el videojuego "*FarCry*". Hace especialmente en el tratamiento de la iluminación. Se puede licenciar a través

[13] https://www.unrealengine.com/products/udk
[14] http://www.garagegames.com/products/torque-3d
[15] http://cryengine.com/

de *Steam*, con un coste de aproximadamente 10 euros al mes a fecha de 25 de junio de 2014, aunque también existe una licencia para estudiantes.

1.2.2.5 Unity3D [16]

Motor de videojuegos multiplataforma desarrollado por *Unity Technologies.* Inicialmente se trataba de un motor de videojuegos para *OS X*, pero hoy en día se utiliza principalmente para el desarrollo de videojuegos orientados a dispositivos móviles y juegos web. Se desarrolló bajo C/C++ y es compatible con código C#, JavaScript y BOO. Permite publicar en prácticamente toda las plataformas existentes actualmente, pero pagando la correspondiente licencia. Además, ofrece dos modos de licencia:

- **Unity:** Licencia gratuita orientada a desarrolladores independientes. Permite publicar sin límites para las plataformas de escritorio, *iOS* y *Android.* Existen algunas limitaciones en cuanto a funcionalidad, como por ejemplo, el uso de *sockets.*
- **Unity Pro:** Incluye todas las funcionalidades de Unity. Su precio a fecha de 25 de junio de 2014 varía entre 1500 dólares en un único pago, o 75 dólares al mes.

Este motor es el que ha sido escogido para el desarrollo del proyecto, pues es una opción bastante asequible, cuenta con una documentación clara y sencilla, una comunidad muy activa, y por ser uno de los motores que más se está popularizando en casi todos los estudios de pequeño tamaño.

1.3 Competencias

A continuación se detallan algunas de las competencias cubiertas durante el desarrollo del proyecto.

G1. Poseer y comprender conocimientos en un área de estudio (Ingeniería Informática) que parte de la base de la educación secundaria general, y se suele encontrar a un nivel que, si bien se apoya en libros de texto avanzados, incluye también algunos aspectos que implican conocimientos procedentes de la vanguardia de su campo de estudio.

Este proyecto cubre la competencia descrita al tratarse de un estudio y desarrollo de técnicas propias de distintas ramas específicas de la ingeniería informática, como son la inteligencia artificial, o el desarrollo de videojuegos.

[16] http://unity3d.com/

G3. Reunir e interpretar datos relevantes (normalmente dentro de su área de estudio) para emitir juicios que incluyan una reflexión sobre temas relevantes de índole social, científica o ética.

El completo desarrollo de este proyecto se ha basado en la búsqueda de información sobre distintas técnicas utilizadas en el desarrollo de videojuegos, y, tras su implementación, se concluye con una reflexión sobre el trabajo realizado y la aportación de dichas técnicas tanto al proyecto realizado como en la industria del videojuego.

G4. Transmitir información, ideas, problemas y soluciones a un público tanto especializado como no especializado.

Este documento pretender ser una guía introductoria para toda persona sienta curiosidad hacia el desarrollo de videojuegos, por lo que mantiene una estructura accesible a todos los públicos, sin requerir grandes conocimientos sobre el tema tratado.

N3. Contribuir a la mejora continua de su profesión así como de las organizaciones en las que desarrolla sus prácticas a través de la participación activa en procesos de investigación, desarrollo e innovación.

El presente documento pretende contribuir a la mejora de la industria del videojuego sirviendo como base para aquellas personas que deseen iniciarse al desarrollo de videojuegos. Por tanto, esta competencia queda cubierta por el documento en sí mismo, así como específicamente por los capítulos orientados al desarrollo, especialmente el de "*Redes*" y el de "*Inteligencia Artificial*".

N5. Participar activamente en la integración multicultural que favorezca el pleno desarrollo humano, la convivencia y la justicia social.

Todos los animales aprenden todos los conocimientos que necesitan para su supervivencia a través de juegos. Las personas, no dejamos de ser animales, y gran parte del aprendizaje que realizamos es a través de juegos. Por ello, opino que en el desarrollo de videojuegos existe un importante potencial para transmitir a los jugadores valores y aprendizajes de una manera efectiva. Esta competencia queda cubierta en la naturaleza y filosofías propias del campo de estudio en cuestión.

T8. Conocimiento de las materias básicas y tecnologías, que capaciten para el aprendizaje y desarrollo de nuevos métodos y tecnologías, así como las que les doten de una gran versatilidad para adaptarse a nuevas situaciones.

El que no recuerda su historia está condenado a repetirla. Este proyecto es, en gran parte, un resumen de técnicas y tecnologías usadas dentro del ámbito de la informática, y en concreto en el desarrollo de videojuegos, a lo largo de su historia. Con estas herramientas e ideas conocidas, avanzar hacía nuevas formas de desarrollo se convierte en una tarea mucho más sencilla.

TFG01. Ejercicio original a realizar individualmente y presentar y defender ante un tribunal universitario, consistente en un proyecto en el ámbito de las tecnologías específicas de la Ingeniería en Informática de naturaleza profesional en el que se sinteticen e integren las competencias adquiridas en las enseñanzas.

Por supuesto, el presente proyecto representa un ejercicio de trabajo original que busca aunar esfuerzos entre el desarrollo de videojuegos, ingeniería informática y la divulgación de técnicas y tecnologías relacionadas con el desarrollo de videojuegos. Esta competencia quedará completa en el momento en que se defienda ante el tribunal asignado.

Capítulo 2: Beat Fighters

"Como cualquier programador de juegos sabe,
los 3 tipos básicos de alimento son Fritos, Cheetos y Doritos"
Satoru Iwata , Presidente de Nintendo

2.1. Diseño de juego de Beat Fighters

2.1.1 Ficha del Juego

- **Nombre:** Beat Fighters.
- **Género:** Memoria / Musical.
- **Plataforma:** PC.
- **Modos de Juego:** Un jugador / Multijugador.

2.1.2 Descripción

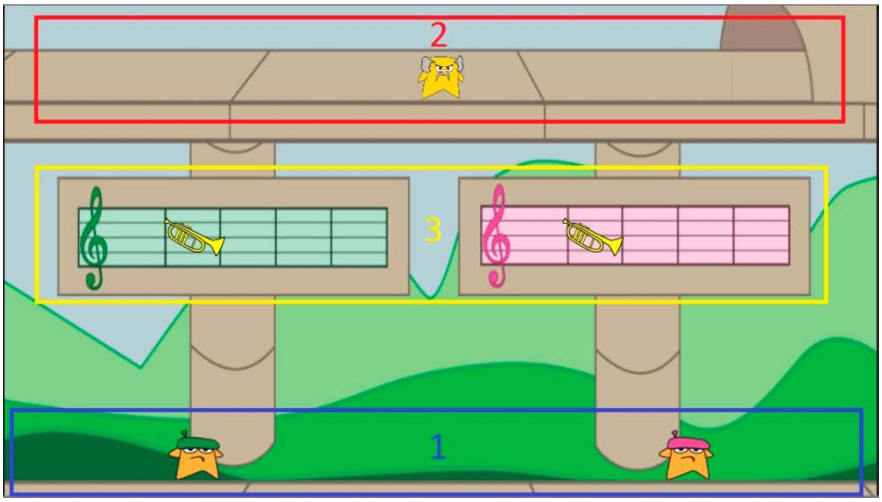
Beat Fighters es un juego de **memoria** con algún toque **musical**. El jugador controlará un personaje con el **objetivo** de **rellenar una partitura** utilizando una serie de **instrumentos** que son lanzados desde la parte superior del mapa. El orden en que deben colocarse estos instrumentos en la partitura se muestra al principio de la partida, por lo que el jugador deberá recordar el orden en el que debe colocar los instrumentos. Todo esto deberá hacerse lo más rápido posible, pues en todo momento nos estaremos enfrentando a otro jugador con los mismos objetivos que el jugador.

2.1.3 Mecánicas

2.1.3.1 El Mundo

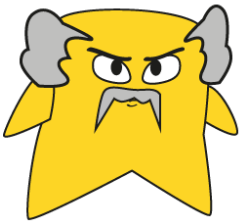
El juego se desarrollará en un mundo 2D. La cámara es fija, y siempre muestra la totalidad del mapa. El mundo se divide en tres partes:

- **Zona inferior:** En esta zona es donde se encuentran los jugadores. Supone el suelo, y es donde caen los instrumentos y por donde los jugadores se mueven.
- **Zona superior:** Esta zona pertenece a la figura de "El Profesor". Esta mecánica se explica más adelante.
- **Zona intermedia:** Son los pentagramas. Los instrumentos deben colocarse aquí en el orden correcto.



División del mundo en Beat Fighters

2.1.3.2 El Profesor



Es un personaje no jugador (NPC). Este personaje será el que, desde la parte superior, deje caer los distintos instrumentos que los jugadores utilizaran para resolver el puzle. Estos instrumentos se generarán en la parte derecha de su zona de acción, de forma que los jugadores no verán el momento en que aparece el nuevo instrumento. Se generarán de forma aleatoria, con mayor probabilidad de los instrumentos que más aparecen en la solución. A continuación se moverá hasta cualquier posición de su zona de acción, desde donde dejará caer el instrumento a la zona de los jugadores. Entre punto y punto, el profesor esperará durante unos instantes.

2.1.3.3 Instrumentos y Pentagrama

En el juego existirán 4 instrumentos distintos, representados por un sprite y un sonido únicos. Estos instrumentos serán: **Caja, Guitarra, Violín y Trompeta.** Deben ser colocados en los pentagramas. Los instrumentos caerán desde la zona de "El Profesor" hasta donde están los jugadores. Si no son recogidos dentro de un determinado tiempo, son descartados automáticamente.

Cada pentagrama se divide en 5 **compases**. En cada compás cabe un solo instrumento. Si se lanza un segundo instrumento sobre un compás ya ocupado, el nuevo instrumento reemplazará al que ya estaba. Si los cinco instrumentos son correctos, el jugador cuyo pentagrama ha sido completado ganará automáticamente. Los pentagramas se corresponden con los jugadores mediante un código de colores.

2.1.3.4 Jugadores

El jugador estará representado por un avatar como el de la derecha. Para diferenciarse, se utilizarán distintos colores para la boina, siendo los posibles colores **verde** y **rosa**, coincidiendo con los colores de los pentagramas.

Los jugadores podrán desplazarse a izquierda y derecha. Al entrar en contacto con un **instrumento**, lo recogerán automáticamente, y podrán elegir entre dos acciones, **descartar** el instrumento lanzándolo hacia la pantalla, o, en su lugar, intentar **colocar** el instrumento en uno de los pentagramas. Podrán colocar instrumentos en cualquier pentagrama, no necesariamente en el propio.

2.1.4. Modos de Juego

En Beat Fighters existirán dos modos de juego:

- **Modo Un Jugador:** El jugador se enfrentará a una inteligencia artificial.
- **Modo Dos Jugadores:** El jugador se enfrentará a otro jugador humano, ya sea en el mismo dispositivo o a través de la red.

2.1.5. Controles

Los controles para el jugador 1 son los siguientes:

- **A - D:** Moverse a Izquierda / Derecha
- **W:** Lanzar un instrumento recogido hacia los pentagramas.
- **S:** Descartar un instrumento recogido.

Para el jugador 2, en el modo de juego en el mismo teclado, son los siguientes:

- **Flecha Izquierda – Flecha Derecha:** Moverse a Izquierda / Derecha
- **Flecha Arriba:** Lanzar un instrumento recogido hacia los pentagramas.
- **Flecha Abajo:** Descartar un instrumento recogido.

Estos son los controles que han sido elegidos para los dos jugadores cuando usan el mismo teclado. La elección de los mismos se debe a dos razones:

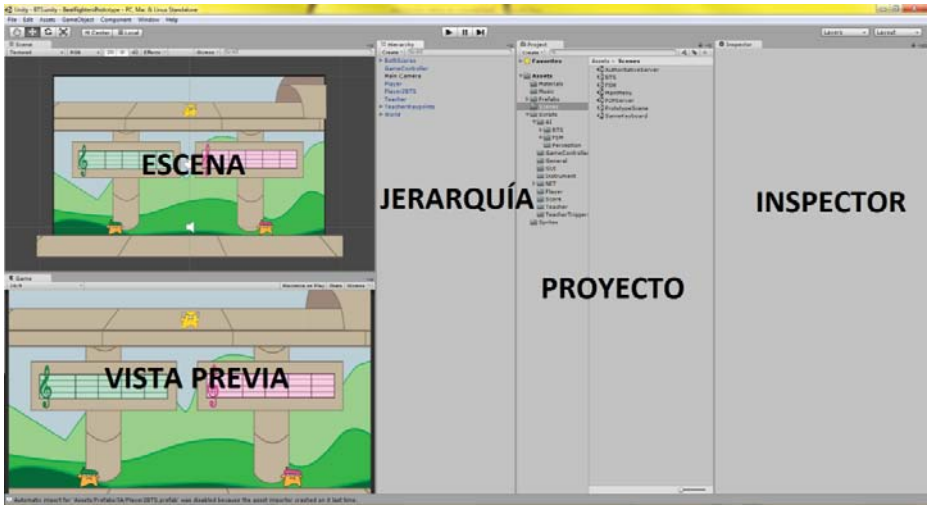
1. **Son las teclas más alejadas del teclado**, permitiendo a los jugadores jugar sin molestarse el uno al otro.
2. **Son las teclas más utilizadas en la mayoría de juegos.** Cualquier jugador automáticamente utilizará estas teclas por mera costumbre.

2.2. Implementación del esqueleto de Beat Fighters

En las siguientes líneas describiré algunos de los aspectos más interesantes del desarrollo de la parte común a todas las próximas implementaciones de Beat Fighters. En primer lugar introduciré algunos de los conceptos más interesantes de Unity3D y a continuación, explicaré como fue el desarrollo de la primera etapa de Beat Fighters.

2.2.1 Introducción a Unity3D

Empezaremos describiendo la interfaz de Unity, que podemos en la siguiente figura:



Interfaz de Unity3D

La **escena** es el mundo virtual donde colocamos todos los elementos que participan el en juego. En la **jerarquía** podemos ver la lista de objetos ya contenidos en la escena que pueden organizarse de forma jerárquica. En la ventana de **proyecto** tenemos todos los elementos que ya se han desarrollado, desde código hasta el contenido artístico, que pueden ser añadidos a la escena. Para ser añadidos a la escena tienen que ser o formar parte de un **GameObject**.

Todos los objetos que contiene la escena son del tipo **GameObject,** la clase más utilizada en Unity y que a su vez hereda de la clase **MonoBehaviour.** Un GameObject es, en realidad un contenedor de **Componentes.** Cada componente aporta al GameObject una nueva funcionalidad. Los scripts que se desarrollen son tratados como componentes, por lo que para su funcionamiento sólo es necesario añadirlo en el GameObject correspondiente. Todos los GameObjects, incluso los GameObjects vacíos tienen en componente **Transform**, que permite controlar la posición, rotación y escala del mismo dentro de la escena.

Siempre que creamos un nuevo script en Unity, automáticamente genera el siguiente código:

```
using UnityEngine;
using System.Collections;

public class ScriptDeEjemplo : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

En este código autogenerado podemos ver dos de los métodos de uso más frecuente en Unity, heredados de MonoBehaviour y que tienen mucho que ver con el ciclo de vida de un componente. A continuación listamos algunos de los métodos similares más interesantes:

- **Awake:** Es llamado cuando la instancia del componente es cargado. Su uso suele ser el de inicializar variables o estados del juego antes de que dé comienzo el mismo. Es decir, es muy similar a un constructor. No es posible saber durante la carga de los componentes, el orden en que estos serán cargados.
- **Start:** Una vez se ha llamado al método Awake de todos los componentes (es decir, todos están cargados) se llama al método Start. Solo es llamado si el componente se encuentra activo, y será llamado (al igual que Awake) una sola vez. Su uso más común es para inicializar referencias hacia otros componentes.
- **Update:** Método llamado en cada frame siempre y cuando el componente esté activo. Es donde se implementa la mayor parte de comportamientos de los GameObjects. Será llamado para todos los componentes activos, y siempre lo más rápidamente posible. El tiempo transcurrido entre dos llamadas a este método se puede consultar en el atributo de la clase *Time.deltaTime.*
- **LateUpdate:** Al igual que Update se ejecuta cada frame si el componente está activo. Se ejecutará cuando todos los updates se hayan ejecutado. Es útil si se necesita esperar por algún dato que se calcula durante el update, o simplemente para establecer un orden de ejecución de los mismos.
- **FixedUpdate:** Similar a los anteriores, pero en este caso el tiempo entre ejecuciones es fijo, por lo que podría incluso ejecutarse varias veces en un mismo frame, si este se demorara demasiado en calcularse. Unity realiza sus cálculos físicos en este método, por lo que si el GameObject incluye el componente *Rigidbody (*componente que permite aplicar fuerzas y velocidades sobre un objeto) debe usarse este método.

Para más información sobre Unity, en la página web http://unity3d.com/learn podemos encontrar documentación, referencia de su API, tutoriales, etc.

2.2.2 Implementación

Para la realización del esqueleto de Beat Fighters me propuse realizar pequeñas iteraciones orientadas cada una a la implementación de una nueva funcionalidad. Así, dividí el juego en seis apartados, que son los que siguen:

1. Jugadores y Movimiento.
2. Profesor y Generación de Instrumentos
3. Instrumentos
4. Pentagramas
5. GameController
6. GUI.

A continuación analizaremos cada iteración brevemente.

2.2.2.1 Jugadores y Movimiento

El primer paso es tener algo que se mueva por la pantalla respondiendo a los eventos de teclado. Para ello se desarrollaron tres componentes: *KeyListener, PlayerMovement y InstrumentController.*

El componente ***KeyListener*** es el encargado de leer del teclado. Según la pulsación realizada, este componente se comunicará con el componente correspondiente, ya sea *PlayerMovement* o *InstrumentController.* Unity3D facilita el trabajo de personalizar los controles mediante la clase Input. Permite dar nombre a las pulsaciones y luego buscar por ese nombre, por lo que si se desea cambiar la tecla relacionada a un movimiento, no es necesario cambiar el código. Este componente es independiente de los componentes que convierten las pulsaciones en acciones, por lo que, si se desease cambiar el sistema de entrada por, por ejemplo, una pantalla táctil, sólo habría que realizar cambios aquí (o crear un nuevo componente y sustituirlo).

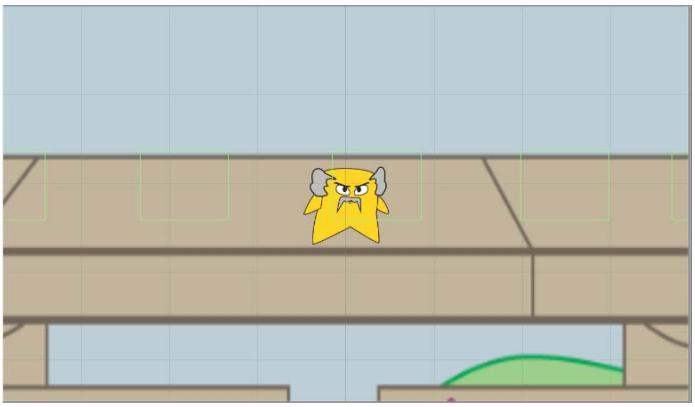
El componente ***PlayerMovement*** tiene la responsabilidad de aplicar el movimiento correspondiente sobre el personaje. Para ello, mantiene una referencia con el componente *RigidBody*, al que asigna la velocidad que se corresponda con el movimiento a realizar. También controla la orientación del personaje. Es independiente del componente *KeyListener*.

Por su parte, el ***InstrumentController*** se encarga de la relación entre el personaje y el instrumento que tenga cogido. Guardará una referencia al otro GameObject que representa al instrumento y, cuando sea necesario, le informará de qué acción debe ejecutar.

2.2.2.2 Profesor y Generación de Instrumentos

Para el movimiento del profesor se utilizó una solución muy similar a los jugadores, que se encuentra en el componente ***TeacherMovement.*** Pero en este caso no tenemos un *KeyListener* que nos indique cuando debemos cambiar el movimiento. Para ello, se han implementado una serie de ***triggers***.

Un trigger es, literalmente, un disparador. Se trata de un GameObject sin componente gráfico, pero con componente *collider.* Este componente permite al GameObject recibir y calcular colisiones físicas. Si se indica que el *collider* debe comportarse como un trigger, este no presentará oposición al objeto que colisione con él, no lo detendrá. Sin embargo, una serie de llamadas a métodos entrarán en acción, siendo estos métodos *OnTriggerEnter, OnTriggerStay* y *OnTriggerExit.* En nuestro caso usaremos el método *OnTriggerEnter*, que es llamado al iniciarse la colisión.



El Profesor rodeado de Triggers

Cuando el trigger se dispara, informa al Profesor que ha llegado a un posible destino. Será el propio profesor el que decida si se trata del punto objetivo, y, en caso afirmativo, se preparará para ir al siguiente destino. Este comportamiento puede verse en el siguiente fragmento de código:

```
private void PointReached(Transform other)
{
    if (actualWaypoint.Equals(other))
    {
        StopMovement();
        instrumentAttached.ThrowInstrument();
        Invoke("NextWaypoint", timeBetweenPoints);
    }
}
```

El método Invoke sirve para hacer una llamada a una corrutina. La rutina en cuestión se identifica mediante su nombre, y se puede asignar un tiempo de espera antes de hacer la llamada al método. En este caso, se utiliza para generar un pequeño retardo entre los puntos por los que se mueve el profesor.

Hay dos tipos de triggers distintos en el camino del Profesor. Uno de estos tipos es el que cumple el comportamiento descrito ya y utiliza el componente ***WaypointTrigger***. El otro, además de este componente, utiliza también el componente ***GenerateTriggerBehaviour.*** Este componente indica al juego que debe generarse un nuevo instrumento y colocarlo en el profesor. Para la creación de nuevos GameObjects se utiliza el método Instantiate, como puede verse en el siguiente fragmento de código:

```
public class InstrumentGenerator : MonoBehaviour {
    ...
    public GameObject GenerateInstrumentAtPosition(Vector3 pos)
    {
        return (GameObject)Instantiate(ChooseRandomInstrumentToGenerate(), pos,
                                       Quaternion.identity);
    }
    ...
}
```

Los parámetros segundo y tercero indican la posición y rotación del nuevo objeto instanciado. Por su parte, el primer parámetro es el GameObject que se desea instanciar. Para ello, se utiliza otra de las características de Unity: los **prefabs**.

Un prefab es un GameObject prefabricado. Se guarda en la jerarquía de proyecto, como cualquier otro recurso del juego. Su uso permite tiene dos beneficios directos: la **reutilización** de un mismo GameObject con facilidad, y que, con solo **modificar** el prefab todos sus clones quedan modificados.

2.2.2.3 Instrumentos

Hasta este punto del desarrollo se habían estado usando *"instrumentos"* falsos para comprobar que los comportamientos implementados hasta el momento funcionasen. Para los instrumentos reales, se han desarrollado cinco scripts, combinados con el uso de un *collider* y un *rigidbody.* Brevemente, son los siguientes:

- **LifeTime:** Controla el tiempo de vida de un instrumento. Si el tiempo de vida es superado, el instrumento se descarta automáticamente.
- **DestroyInstrument:** Encargado de destruir la instancia cuando esté fuera de la pantalla.
- **InstrumentCollision:** Se encarga de manejar las colisiones con el instrumento. Siempre intentará engancharse al objeto con el que ha colisionado. Si no puede hacerlo, se descarta automáticamente.
- **InstrumentController:** Se encarga del movimiento del instrumento. En las etapas del ciclo de vida (del que hablaremos a continuación) en que el instrumento deba moverse de forma controlada, este script es el que toma el control. También es aquí donde se controla el factor de escala para cuando el instrumento está siendo descartado.
- **InstrumentLifeCycle:** Define y controla el ciclo de vida de un instrumento. Este ciclo se divide en 6 etapas. Su flujo puede verse en la imagen a continuación:

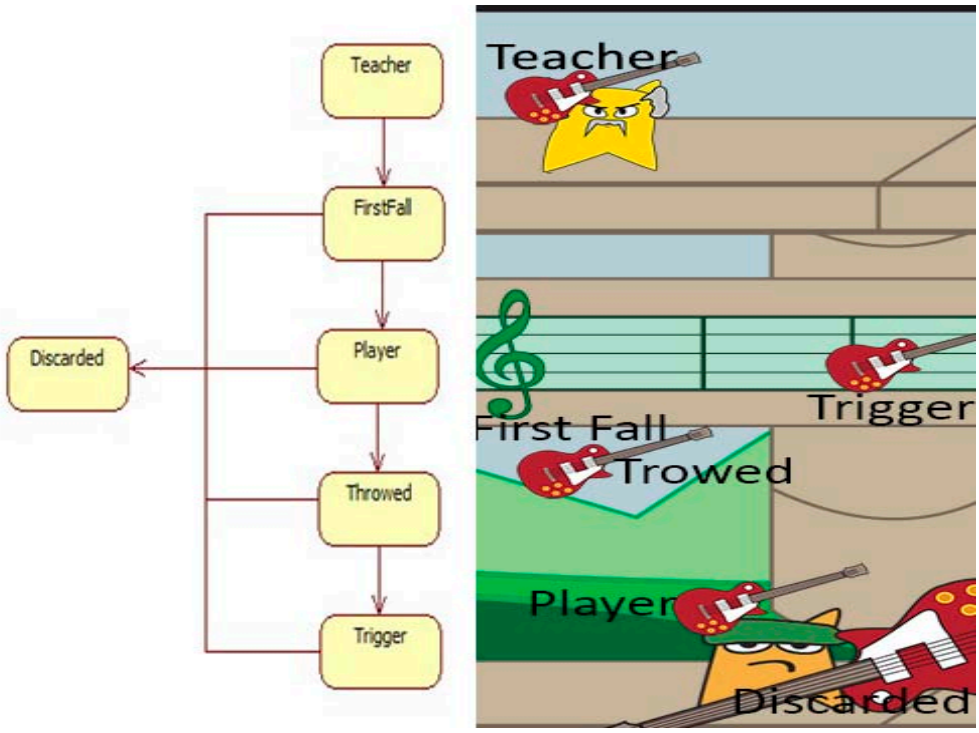


Diagrama Ciclo de Vida de un instrumento

Como puede observarse, según el momento del ciclo de vida en que se encuentre el instrumento, el comportamiento será distinto. De esta manera, controlamos cómodamente que componentes del instrumento queremos tener activos en un cierto momento de juego.

2.2.2.4 Pentagramas

A estas alturas, solo nos faltan los pentagramas para tener todos los elementos que componen el juego. Un pentagrama es, por definición, la reunión de cinco líneas y cuatro espacios donde se escribe la música. Además, cada pentagrama está dividido en compases. Para nuestra implementación mantuvimos esta diferenciación y jerarquía.

Los compases en Beat Fighters son triggers. Cuando un instrumento colisiona con uno de estos triggers, el compás lo atrapará. El instrumento permanecerá en el compás hasta que otro instrumento haga dispararse al trigger. Este comportamiento se encuentra en el script ***ScoreTriggerBehaviour.***

Por su parte, el pentagrama en si lo único que necesita hacer es guardar referencias a todos los compases que lo componen. El único servicio que ofrecerá al exterior será la posibilidad de consultar la "partitura" escrita (los instrumentos que se encuentran en el pentagrama y su orden). Este comportamiento está totalmente ligado con la siguiente iteración y puede encontrarse en el script ***ScoreManager***.

2.2.2.5 GameController

La entidad GameController se encuentra, de forma implícita o no, en todos los juegos. Se encarga de controlar el flujo de juego, es decir, cuando y como empieza el juego, cuando acaba, y cuando se producen los eventos especiales, si los hubiera. También se encargaría de controlar las partidas guardadas, pausas, etc…

En el caso de Beat Fighters tiene tres funciones principales. **Generar la melodía** objetivo de la partida, **poner en marcha al profesor** para que el juego pueda empezar, y **controlar la victoria** para que el juego pueda acabar. El flujo de juego puede verse claramente en el siguiente fragmento del script ***LevelController***.

```
public class LevelController : MonoBehaviour {
    ...
    void Start()
    {
        InitScore();
        InitGenerator();
        FindScoreManagers();
        this.GetComponent<StartLevelManager>().StartGame(score);
    }

    public void CheckVictory()
    {
        if (CheckWinCondition(greenScore) ||
            CheckWinCondition(pinkScore))
        {
            EndGame();
        }
    }
    ...
}
```

En el método Start, lo primero que se hace es iniciar la "partitura" objetivo del nivel (**InitScore**). A continuación se inicializa el generador de instrumentos para poder instanciar instrumentos en cualquier momento y se crean las referencias a los dos pentagramas (**InitGenerator** y **FindScoreManagers**). Finalmente llama al método **StartGame** que se encuentra en el componente ***StartLevelManager***. Este método es el encargado de iniciar la partida. Reproducirá la melodía mostrando los instrumentos en orden para dar *feedback* visual al jugador, y finalmente pondrá en marcha al profesor.

En este punto están todos los elementos necesarios para jugar una partida de Beat Fighters contra otro rival humano en el mismo ordenador.

2.2.2.6 Interfaz gráfica de Usuario (GUI)

Unity3D proporciona servicios en su API para pintar una interfaz de usuario. Estos servicios se encuentran en la clase GUI, y permite dibujar botones, cuadros de texto, etiquetas… En Unity3D, todo el proceso de pintado se produce en el método ***OnGUI***. Este método hay que usarlo con precaución, pues es llamado varias veces por frame, pudiendo provocar grandes pérdidas de rendimiento.

Como en Beat Fighters va a haber, presumiblemente, una GUI diferente en cada una de las etapas, hemos optado por crear una clase básica, ***GUIBasics,*** que se encarga de obtener y mantener las medidas de la pantalla y pintar los elementos que van a estar presentes en todas las pantallas. De esta clase heredarán las GUI's específicas de cada apartado del juego, aprovechando las medidas ya tomadas por esta GUI.

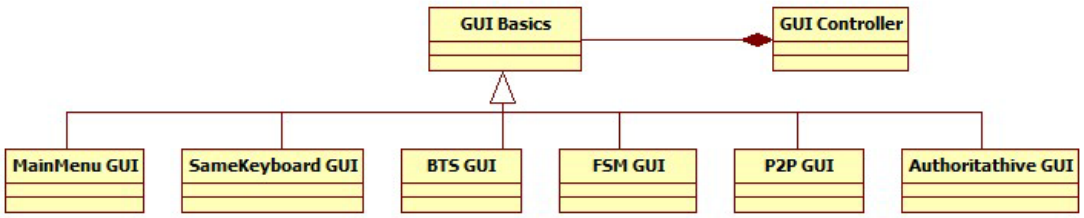


Diagrama de clases para la GUI. Incluye todas las clases que se implementaron a lo largo del desarrollo.

Para controlar cual GUI debe cargarse en cada escena, se implementó un ***GUIController***. Este controlador mantiene el patrón singleton (sólo puede existir una instancia) a nivel de su GameObject. Sólo puede existir en el juego una sola instancia del objeto que se va a encargar de sostener la GUI. Cada vez que se carga una nueva escena, este controlador comprueba a que escena se encuentra el juego, y cargará la GUI correspondiente.

Capítulo 3: Inteligencia Artificial

"Lo siento, Dave. Me temo que no puedo hacer eso"
HAL 9000 , El Ordenador de 2001: Odisea en el Espacio

3.1. Introducción

¿Qué es la inteligencia? Quizás, antes de poder hablar correctamente sobre inteligencia artificial y más concretamente sobre su uso en videojuegos, sea interesante hablar del propio concepto de inteligencia. Podríamos definir la inteligencia como un conjunto de propiedades de la mente, tales como la capacidad de hacer un plan, o resolver problemas. Incluso podríamos definirlo de una forma más simple y decir que la inteligencia es la habilidad de tomar la decisión correcta dada una serie de entradas y una variedad de posibles acciones.

Si usamos esta última definición, nos damos cuenta de que este comportamiento no se ve solamente en seres humanos, sino también en animales. Sin embargo, es obvio que la inteligencia mostrada por los seres humanos es mayor. Por ejemplo, los seres humanos tenemos la capacidad de comunicarnos mediante el lenguaje, capacidad que comparten algunos animales, pero a menor escala. Al igual que esto, el ser humano es capaz de resolver problemas, al igual que algunos animales, sólo que el ser humano es capaz de resolver problemas más complejos. Nos damos cuenta entonces que el ser humano posee unas habilidades de las cuales el resto de animales poseen una capacidad algo inferior.

Cuando hablamos de inteligencia aplicada a seres artificiales podríamos aplicar la misma analogía. Podemos crear un programa que sea capaz de jugar al ajedrez como un campeón mundial, pero este programa no será capaz de jugar a otro juego, como podría ser un simple parchís. Desde esta perspectiva, la más inteligente de las aplicaciones podría serlo desde cierta perspectiva, pero desde otro ángulo ser tan simple como el menos inteligente de los animales.

Por tanto, podríamos decir que la inteligencia artificial consiste en conseguir que una máquina sea capaz de realizar razonamientos de los cuales las personas y animales son capaces. Ya hemos conseguido que un programa tenga habilidades sobrehumanas en la resolución de ciertos tipos de problemas, como por ejemplo cálculos matemáticos, ordenación… Sin embargo, en tareas que podríamos considerar triviales, como reconocer rostros familiares, o la toma de decisiones, las máquinas aún no son tan capaces como el resto de organismos vivos.

Muchos de los avances en inteligencia artificial están motivados por estudios filosóficos. Buscan entender la naturaleza del pensamiento. Otras veces, están motivados por el campo de la psicología, buscando entender las mecánicas de los procesos mentales. Por otra parte, en ingeniería está motivado por intentar dar a las máquinas y al software la capacidad de resolver tareas como una persona lo haría. Este es el campo que más interesa en el desarrollo de videojuegos, pues, buscamos que los distintos personajes con los que el jugador interactúa dentro del juego tengan un comportamiento cercano al de un ser humano.

3.2. Un poco de Historia

La capacidad para pensar de una máquina no es precisamente un concepto novedoso. Ya en los mitos griegos puede avistarse un acercamiento a esta idea en el mito de Talos, un hombre de bronce creado por Hefesto que protegía a la bella Europa en Creta.

A su vez, los filósofos buscaban respuestas a preguntas como "¿De dónde proviene el pensamiento?" o "¿Qué diferencia a un cadáver del hombre que fue antes?". En la Europa victoriana empieza a existir un gusto por los robots y seres no del todo humanos dotados de inteligencia, siendo pionera la novela *"Frankenstein"*.

En la década de los 40 podemos encontrar los primeros intentos de realizar tareas humanas mediante máquinas programadas. Esto se debe a la necesidad, durante la Segunda Guerra Mundial, de descifrar los códigos enemigos y realizar los cálculos necesarios para el desarrollo de las armas nucleares que posteriormente serían utilizadas en el conflicto. Pioneros de la computación como Turing o von Neumann fueron también pioneros en el campo de la inteligencia artificial. A Turing se le considera uno de los padres de la inteligencia artificial, en parte por ser el creador del "Test de Turing", en el que postula que si el comportamiento de una máquina es indistinguible del de una persona, entonces podemos considerar que la máquina se comporta como tal.

Dramatización del Test de Turing

En esta línea, Turing propuso la idea de una "máquina niño". Consideró que sería posible obtener una inteligencia adulta a partir de una base de conocimiento y el aprendizaje. Este concepto evolucionaría en lo que hoy conocemos como sistemas expertos, que a partir de una base de conocimiento son capaces de aplicar reglas a dicho conocimiento para resolver problemas u obtener más conocimiento.

En 1956 Allen Newell, junto con Herbert Simon y J.C. Shaw desarrollaron el primer programa considerado de inteligencia artificial. Se trataba de un programa capaz de resolver ecuaciones. Ese mismo año, durante la conferencia de Dartmouth, sería acuñado el término de Inteligencia Artificial. Sería pronunciado por primera vez en los labios de John McCarthy y lo definió como "La ciencia e ingeniería de fabricar máquinas inteligentes". Fue también el creador del lenguaje LISP, y ganador del premio Turing en 1971.

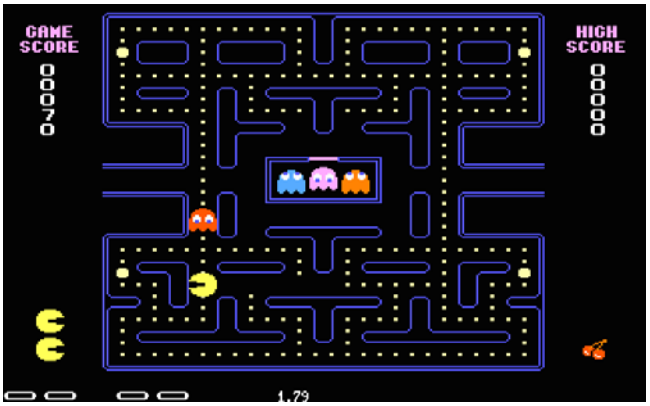
Hasta la década de los 70, los estudios en inteligencia artificial habían generado un interés considerable, y el desarrollo de nuevas técnicas como las redes neuronales habían generado bastante excitación en la comunidad. Sin embargo, es a raíz del artículo "Perceptrones" de Marvin Minsky y Seymour Papert, en el que demostraban las limitaciones de los perceptrones mono capa en especial enfrentados a problemas no linealmente separables, que se produce un abandono de las investigaciones en el campo de la inteligencia artificial. Solamente los sistemas expertos continuaron su avance, produciéndose el nacimiento del lenguaje Prolog.

Blade Runner, Un ejemplo de la Inteligencia Artificial en la cultura popular

A mediados de la década de los 80 el interés por la inteligencia artificial parece volver a florecer, pero desde otro punto de vista. En lugar de intentar crear máquinas inteligentes, busca centrarse en metas específicas, intentando mejorar los puntos débiles del campo. Además, se empiezan a seguir nuevas aproximaciones inspiradas en la biología, abandonando en parte la visión simbólica utilizada hasta el momento. Aparecen los algoritmos genéticos, que al ser combinados con las redes neuronales dan unos resultados extraordinarios.

En la década de los 90, el campo de la inteligencia artificial ha recobrado fuerza y empiezan los estudios de la "vida artificial". Este nuevo campo nace para simular grupos de formas de vida, y así poder analizar conceptos como la evolución o la aparición del lenguaje. Consecuencia de esto es la aparición del enfoque de sistema dentro de la inteligencia artificial. Este enfoque significa simplemente que un algoritmo debe ser usado dentro de un contexto. Por tanto, el algoritmo se ve beneficiado de un sistema desde el que poder recibir los inputs correctos, y es capaz de afectar al mundo de alguna manera.

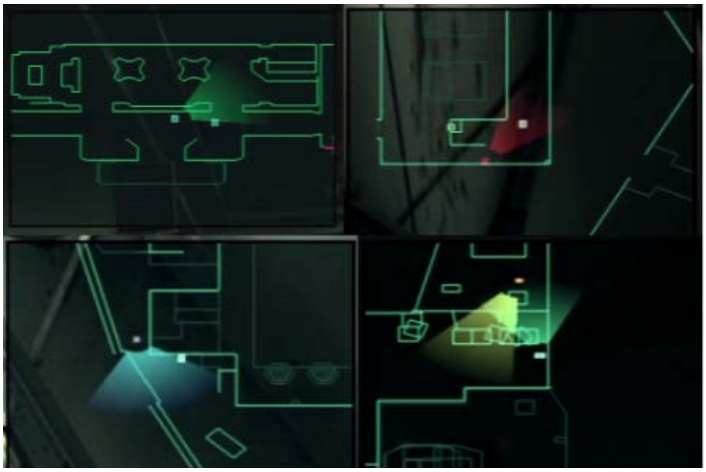
Los estudios en inteligencia artificial siempre estuvieron estrechamente ligados a la resolución de juegos. Ya en 1950, Claude Shannon propuso que el ajedrez era en realidad un problema de búsqueda. Sin embargo, no sería hasta 1979, con la aparición de Pac-Man, que tendríamos la oportunidad de ver el primer videojuego con una inteligencia artificial notable. Antes, se habían visto aproximaciones como el Pong, o en Space Invaders, aunque en el primer caso el rival se dedicaba simplemente a seguir a la pelota arriba y abajo, y en el segundo a moverse siguiendo un patrón predefinido.



PAC-MAN

Sin embargo, en Pac-Man, los jugadores podían ver como los cuatro fantasmas parecían confabular para rodear al jugador (siempre y cuando este no hubiera recogido una de las pastillas grandes). Cada fantasma tenía su propia personalidad, existiendo uno que se dedicaba a perseguir al jugador mientras otro intentaba rodearlo. Para la realización de este sencillo algoritmo se utilizó la técnica de máquinas de estado. Otros juegos, como Golden Axe (1987) utilizaron también esta técnica.

No sería hasta la década de los 90, con la aparición de las primeras consolas y mejores procesadores, que la inteligencia artificial en videojuegos evolucionaría. GoldenEye 007 introdujo un nuevo concepto: un sistema de percepción. Los enemigos eran capaces de verse entre sí, e incluso de ver a un compañero caído. Esta mecánica fue altamente aprovechada por Metal Gear Solid, que lo incorporó como una de sus mecánicas principales, siendo mostrado el cono de visión de los enemigos en el minimapa.



Conos de Visión en Metal Gear Solid

Es también durante la década de los 90 que empiezan a aparecer los primeros juegos de estrategia en tiempo real. Dune II y Warcraft se erigen como los dominadores del género. Algoritmos de búsqueda de caminos y los aspectos de representación del terreno ganan en importancia dentro de este género.

La inteligencia artificial en videojuegos empieza a convertirse en parte principal de los mismos, como demostraron en el año 2000 Los Sims y Black and White. El primero, un simulador de vida humana, destaca por el uso de un sistema de “puntos calientes”. El

personaje central desconoce que objetos debe utilizar para cubrir sus necesidades, son los propios objetos los que aportan tal información. Además, es el propio objeto el que indica al personaje como debe usarlo.

Por otra parte, en Black and White, un simulador de dios al más puro estilo Populous, se incluye una nueva mecánica. Debemos criar a una bestia, que aprenderá dependiendo de cómo reaccionemos a sus acciones. Además, este juego incluye también conceptos de estrategia en tiempo real, algoritmos genéticos y árboles de decisiones. Es considerado uno de los videojuegos más influyentes de la historia en el campo de la inteligencia artificial [6].

Black & White ¿Cómo criarás a tu mascota?

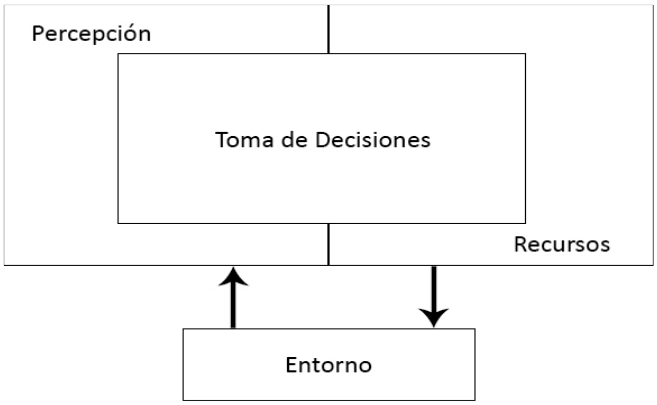
El género de los shooters en primera persona también ha hecho su aportación al campo de la inteligencia artificial. Halo, en 2001, incorpora por primera vez enemigos que se ven afectados por lo que les pasa a sus compañeros, llegando incluso a huir cuando su líder es abatido. Sustituye las máquinas de estados por árboles de comportamiento, técnica que se haría especialmente popular a partir de Halo 2, y que es actualmente una de las técnicas más populares.

Por su parte, F.E.A.R en 2005 realizó un gran esfuerzo en la inteligencia táctica de los enemigos. Estos son capaces de usar el entorno con inteligencia, buscando puntos de cobertura, lanzando granadas a través de ventanas… Pero lo más interesante es la relación entre ellos, siendo capaces de realizar fuego de cobertura o maniobras de flanqueo.

3.3. Inteligencia Artificial en Videojuegos

3.3.1 Arquitectura

Cuando desarrollamos el módulo de Inteligencia Artificial, debemos detenernos a pensar que tipo de aproximación vamos a utilizar. En videojuegos, buscamos que las entidades controladas por la máquina (NPC) reaccionen de forma realista ante los eventos del juego, como si de una persona se tratase. Necesitamos, por tanto, utilizar un enfoque de sistema. Nos basaremos en el descrito en *Artificial Intelligence. A Systems Approach.* [3].



Enfoque de Sistema para la IA.

Como se puede observar en el diagrama anterior, la toma de decisiones depende estrechamente de como la entidad observa el mundo. Se vuelve crítico, por tanto, incluir algún sistema de **percepción** que permita al NPC obtener información de su alrededor.

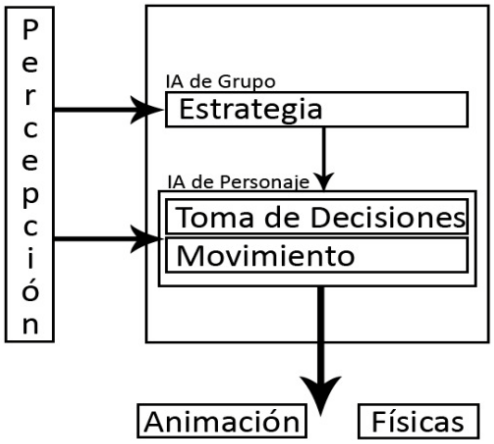
Por otra parte, vemos también que la toma de decisiones afecta, a su vez, al estado del entorno que rodea al NPC. Si decide moverse, disparar, saltar… esta decisión deberá reflejarse de alguna manera en el mundo. Deberá por tanto, tener acceso a los servicios necesarios para controlar al NPC, a efectos de poder llevar a cabo las decisiones tomadas.

Finalmente, el centro del diagrama representa el momento en que la inteligencia del NPC toma el control del procesador y decide qué hacer a continuación. No todas las decisiones serán iguales, y podemos distinguir dos tipos de decisiones: las decisiones de **grupo** (estrategias) y las decisiones **individuales**. Las decisiones individuales pueden verse afectadas por la estrategia, y, además, debe responder a dos preguntas. Qué va a hacer, y cómo lo va a hacer. La primera pregunta es puramente intelectual, mientras que la segunda se correspondería con un **movimiento.**

Por ejemplo: en un juego de estrategia en tiempo real ambientado en la Segunda Guerra Mundial, un **pelotón decide** cambiar su formación. Esto sería una decisión **estratégica**. Seguidamente, **cada unidad deberá decidir** cuál será su nueva posición en la formación. Esta sería la decisión **individual**. Una vez ha decidido su nuevo puesto dentro del pelotón, debe **decidir cómo llegará** a esta nueva posición, es decir, que camino tomará. Esta sería su decisión de **movimiento**.

Todo este proceso descrito anteriormente necesita apoyarse del sistema de percepción. Los NPC's necesitan saber dónde están sus compañeros, si una posición es válida para moverse hasta ella, si hay obstáculos en el camino…. No sólo esto, sino que, para poder llevar a cabo la decisión tomada, la entidad deberá desplazarse, cambiando su animación y su transformación en el mundo.

Tomando en cuenta estas consideraciones, utilizaremos la siguiente arquitectura:



Arquitectura de la IA

No todos los juegos necesitarán todas las divisiones de la toma de decisiones. Por ejemplo, en el ya mencionado Pac-Man, aunque da la sensación de que los distintos fantasmas trabajan en equipo, no tiene una inteligencia de estrategia que la soporte, en realidad, cada uno hace la guerra por su lado. Por otra parte, un juego al estilo del Risk, sólo tomará decisiones estratégicas, obviando toda la parte individual.

3.3.2 Técnicas

A continuación, explicaré brevemente algunas de las técnicas más populares utilizadas en el desarrollo de inteligencia artificial para videojuegos, prestando especial atención a máquinas de estado y árboles de comportamiento, pues son las técnicas de toma de decisiones que se han implementado en Beat Fighters. Tomamos como base la descripción realizada en *Artificial Intelligences for Games Second Edition*[4].

3.3.2.1 Movimiento: Steering Behaviours

Steering Behaviours no es un algoritmo en sí mismo. Se trata de un conjunto de algoritmos que buscan, mediante el uso de estrategias sencillas, hacer que el NPC se mueva de forma realista. No hace uso de algoritmos de búsqueda de ruta, si no de fuerzas simples como la aceleración, tanto del propio NPC como de sus vecinos. Algunos Steering Behaviours son:

- **Seek (Búsqueda)**: El NPC se moverá en dirección a un objetivo. Para ello, en lugar de modificar la velocidad y rotación directamente para encarar el objetivo, irá modificando su velocidad mediante el cálculo de la aceleración necesaria para alcanzar la posición deseada. Normalmente la velocidad y aceleración máximas son proporcionadas al algoritmo, para lograr mayor realismo, pues si no, el NPC podría aumentar su aceleración indefinidamente.
- **Flee (Huida):** Es el algoritmo opuesto al de Seek. En lugar de intentar acercarse, intentará alejarse del objetivo. Al igual que en el algoritmo de búsqueda, intentará acelerar todo lo posible.

- **Arrive (Llegada):** En el algoritmo de búsqueda, el NPC acelerará todo lo posible. Es un algoritmo interesante si el objetivo está en movimiento y el NPC debe atraparlo a toda velocidad. Sin embargo, en caso de alcanzarlo, se dedicará a orbitar a su alrededor. El algoritmo de arrive busca corregir este comportamiento. Se utiliza un radio, a partir del cual se empieza a frenar. En la distancia máxima de este radio, la velocidad del NPC será máxima, mientras que en el objetivo debe ser cero. Según la distancia, la velocidad es interpolada. Por tanto, observamos que este algoritmo no solo modificará su aceleración para moverse en dirección al enemigo, sino que también tendrá que adaptarse a una velocidad objetivo en cada momento.
- **Pursue (Perseguir):** Se trata de una evolución del algoritmo de Seek para objetivos en movimiento. En lugar de solo observar la posición del objetivo, también analizamos su velocidad. Así, podemos predecir dónde se encontrará y en un cierto espacio de tiempo, y convertir este punto en nuestro objetivo del algoritmo de búsqueda.
- **Flooking (Bandada):** Es, probablemente, el más popular de los Steering Behaviours. Simula el movimiento de una bandada, haciendo uso de otros tres algoritmos más sencillos. Estos son: **Separación, Cohesión y Alineación.** El primero se encarga de mantener a cada miembro de la bandada suficientemente separado de sus vecinos para no colisionar con ellos. Mediante el de cohesión, se busca el centro de masa de la bandada. El NPC intentará aproximarse a él. Moverse en la misma dirección y velocidad que el resto de la bandada es el objetivo del algoritmo de alineación. La resultante de las tres aceleraciones obtenidas por cada uno de estos algoritmos será la que se aplique al NPC en cuestión.

Como puede observarse, mediante esta técnica, podemos obtener movimientos realistas sin necesidad de realizar cálculos complejos. Su flexibilidad, bajo coste y capacidad de combinación para crear comportamientos complejos (como el de Flooking) han provocado que se trate de una de las técnicas más populares en la actualidad.

Ejemplo del algoritmo de Seek

*3.3.2.2 Búsqueda de Caminos: A**

Hablar de búsqueda de caminos y no hablar del algoritmo A* es casi pecado. Fuera de toda duda queda la calidad más que demostrada de este algoritmo, utilizado en prácticamente cualquier videojuego que necesite solucionar el problema de búsqueda de caminos.

Se trata de un algoritmo de búsqueda en grafo informado, que se caracteriza por tener en cuenta, para cada nodo, tanto el valor heurístico del nodo como el coste real del recorrido. Así, el algoritmo A* utiliza una función de evaluación $f(n) = g(n) + h'(n)$. La $g(n)$ representa el coste real del recorrido, mientras que $h'(n)$ representa al valor heurístico. Además, este algoritmo utiliza dos estructuras auxiliares, en forma de lista de prioridad ordenada por el valor de la $f(n)$ de cada nodo. Una de las estructuras representa a los nodos abiertos y la otra a los nodos cerrados. En cada iteración, se expande el primer nodo abierto, y si no es el nodo objetivo, se calculan las funciones de todos sus hijos, que se insertan en la lista de nodos abiertos, mientras que el nodo evaluado pasa al de cerrados. De esta forma se itera hasta encontrar el camino con el menor coste posible.

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Celdas de un camino A*, con su coste.

3.3.2.3 Toma de decisiones: Máquinas de Estados

En un videojuego, un NPC normalmente tendrá un número limitado de acciones. Normalmente estarán llevando a cabo algún tipo de comportamiento hasta que un evento del mundo les haga cambiar de estado. En un shooter por ejemplo, los enemigos podrían permanecer ociosos hasta detectar al enemigo, momento en que se disponen a disparar. Este tipo de comportamientos puede ser modelado fácilmente con máquinas de estados.

Las máquinas de estados son, en realidad, autómatas finitos que continuamente están devolviendo algún tipo de salida. Por tanto una máquina de estados está compuesta por:

Q: Conjunto de Estados.
∑: Alfabeto de Entrada.
δ: (Q x ∑) -> Q Función de transición.
q_0: Estado inicial.
Z: Alfabeto de salida.
λ: Función que determina la salida.

Como podemos observar, disponemos de una serie de **estados**, que serán las posibles acciones que nuestro NPC podrá llevar a cabo. El alfabeto de entrada normalmente se corresponderá con nuestro sistema de percepción, que será el que nos informe de los eventos del mundo, y provocará las **transiciones** hacia los distintos estados. Dependiendo de cómo se obtenga la función de salidaλ, podemos hablar de dos versiones distintas de las máquinas de estado:

- Máquina de Mealy: La salida depende del estado actual y de la entrada.

- Máquina de Moore: La salida depende solamente del estado actual.

Normalmente se usan las máquinas de Moore, pues son más sencillas de implementar, y no siempre dispondremos de una entrada que utilizar para obtener la salida.

Esta técnica ha sido una de las más utilizadas a lo largo de la historia de los videojuegos. Sobre su implementación dentro de Beat Fighters se hablará más adelante.

3.3.2.4 Toma de decisiones: Árboles de Comportamiento

Uno de los problemas de las máquinas de estado es su dificultad para ser manejadas por los diseñadores de juego, debido a que tendrían que manejar conceptos de programación que no deben conocer necesariamente. Los árboles de comportamiento unen ideas de distintas técnicas, como las máquinas de estado, planificadores y scripts, y las componen de manera que sean muy sencillas de entender y manejar por personas sin conocimientos de programación.

Existe una fuerte similitud entre las máquinas de estado y los árboles de comportamiento, pero, en lugar de tener estados, el árbol se compone de **tareas.** Cada tarea realiza una parte del comportamiento del árbol. Se componen de manera jerárquica, por lo que se podría crear herramientas gráficas que permitan a los diseñadores construirlos de forma sencilla.

Cada tarea puede ser todo lo compleja que se desee, pero es preferible que sean sencillas, pues así son más reutilizables. Para obtener comportamientos más complejos, lo ideal es combinar varias tareas, en lugar de hacer una tarea con todo el comportamiento.

Existen dos tipos de tareas:

- Tareas Atómicas:
  - Acciones.
  - Condiciones.
- Tareas Compuestas:
  - Secuencias.
  - Selectores.
  - Paralelas.

Las **acciones** modifican el estado del juego: ejecutan una animación, desplazan al NPC… Normalmente las acciones tendrán **éxito.**

Las **condiciones** consultan el estado del juego. ¿Veo al jugador?, ¿Tengo munición? Son similares a las transiciones de las máquinas de estados, pero no lo son. Las condiciones comparten la interfaz de las acciones, y se combinan con ellas mediante el uso de nodos internos. De esta manera, se evita la explosión de transiciones. Dependiendo del estado del mundo, una condición podrá tener **éxito** o **fracasar.**

Las tareas compuestas forman los nodos internos del árbol. Son los encargados de darle el poder expresivo al mismo.

Las **secuencias** ejecutan todas sus subtareas en orden. Si alguna de sus subtareas fallas, la secuencia falla. Sólo tendrá éxito si todas sus subtareas tienen éxito.

Los **selectores** ejecutan sus tareas en orden. Cuando una de sus subtareas tiene éxito, el selector tiene éxito. Sólo fallarán si todas sus subtareas fallan. Por tanto, proporciona varias maneras de lograr el mismo objetivo. Es importante recalcar que el orden de sus hijos establece su prioridad. Para obtener un comportamiento menos predecible, podría implementarse de una forma no determinista. Al iniciarse el selector, reordenaría a sus hijos de forma aleatoria, desapareciendo así la prioridad implícita del selector.

La tarea **paralela** ejecuta a todos sus hijos al mismo tiempo. Puede implementarse con política de secuencia o con política de selector, según cuando debe considerarse que la tarea ha tenido éxito. Si una tarea paralela finaliza antes que alguno de sus hijos, debe asegurarse de detenerlo. Esta tarea permite al NPC realizar varios comportamientos al mismo tiempo, como podría ser disparar mientras se desplaza buscando una cobertura.

Además, para darle más potencia a las distintas tareas, podemos hacer uso del patrón **decorador**, creando tareas que decoren al subárbol que les sigue, añadiéndole cierta funcionalidad extra que no tuviesen antes. También pueden ser usados para controlar el acceso a recursos compartidos, creando un decorador que funcione como un **semáforo**, por ejemplo.

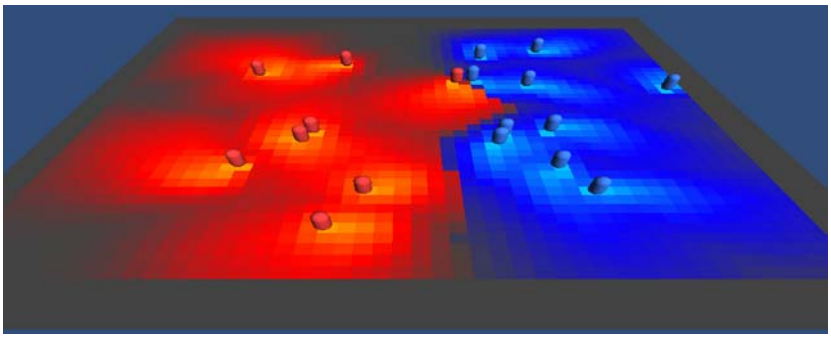
Los árboles de comportamiento se han vuelto especialmente populares en los últimos años, sobre todo después del lanzamiento de **Halo 2**. Sobre sus detalles de implementación hablaremos en profundidad más adelante.

3.3.2.5 Estrategia: Mapas de Influencia

Como ya hemos mencionado antes, la técnica a utilizar depende del juego que se va a desarrollar. Los mapas de influencia son extensamente usados en juegos de estrategia.

Esta técnica discretiza el mundo en celdas, en las que guarda información como puede ser la potencia de combate, el número de soldados, la existencia de recursos… Por tanto, se podrán tomar decisiones como las tomaría un mando militar. Se puede incluso tener varios mapas de influencia distintos con distintos tipos de información: uno con los recursos y las unidades recolectoras, otro con las unidades militares, otro con información sobre el relieve…

A partir de esta información, se establecen los valores de **influencia** de cada celda. Esta influencia se propaga a las celdas vecinas, siempre de forma decrementada. De esta manera, podemos ver al mapa de influencia como una representación discreta del potencial de una celda en cierto sentido. Mediante el análisis del mapa, podemos obtener puntos débiles del contrario, áreas donde tenemos el control estratégico, el frente de combate…

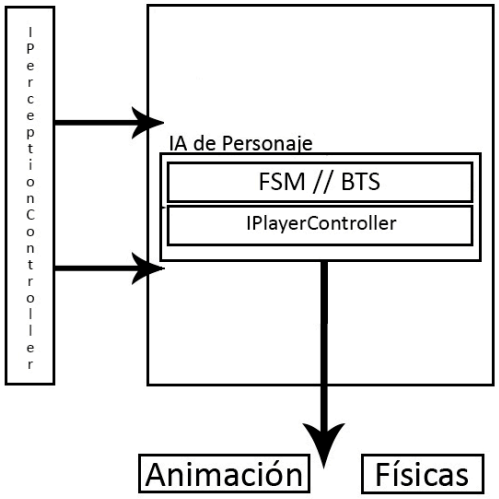


Ejemplo de Mapa de Influencia

3.4. Inteligencia Artificial en Beat Fighters

3.4.1. Arquitectura

En el caso de Beat Fighters, se trata de un juego competitivo uno contra uno. Por tanto, de los módulos descritos en la sección anterior, no necesitaremos el de estrategia. La arquitectura quedará como muestra la siguiente figura:



Arquitectura de la IA en Beat Fighters

Utilizaremos la interfaz **IPerceptionController** para obtener información del mundo, será nuestra comunicación con el sistema de percepción del NPC. La interfaz **IPlayerController** será utilizada para llevar a cabo las acciones necesarias sobre el personaje.

Para los servicios proporcionados por IPlayerController, nos serviremos de los scripts PlayerController e InstrumentControls ya desarrollados para el control de los personajes. Sin embargo, para los servicios proporcionados por IPerceptionController necesitamos desarrollar nuestro módulo de percepción.

3.4.2. Módulo de Percepción

En Beat Fighters, nuestro sistema de percepción tiene que suplir dos necesidades básicas de nuestro NPC. Por una parte, necesitamos desarrollar un **sistema de visión** que le permita al NPC saber dónde están los instrumentos que ya han caído al suelo. Por otra parte, necesita conocer las **posiciones del mundo** interesantes donde colocar los instrumentos, así como el orden en que los instrumentos deben ser colocados. Para este fin, se han desarrollado dos scripts, el primero, implementa un sencillo algoritmo de línea de visión, mientras que el segundo, accede a la información del mundo consultándolo directamente.

3.4.2.1. Línea de visión

El algoritmo de línea de visión es el que permite a nuestro NPC saber dónde están los instrumentos más cercanos que han caído al suelo. Para ello, utilizo una técnica denominada "Ray Casting". Esta técnica consiste en lanzar rayos físicos que colisionan con el componente físico collider de los objetos y nos devuelve información acerca de la colisión, incluyendo al objeto colisionado.

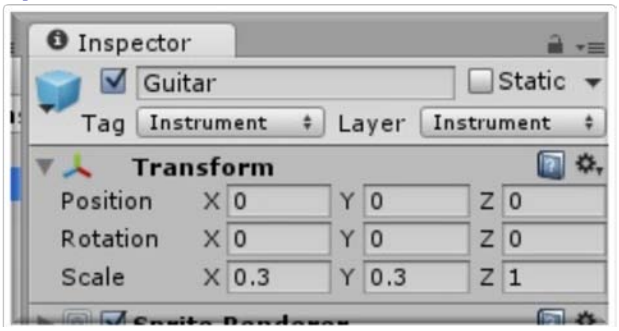
Para evitar que este sentido nos devuelva información acerca de la posición del otro jugador, aprovecharemos el concepto de capas que nos aporta Unity, colocando los instrumentos en su propia capa, y colisionando sólo con esa capa.

```
public class LineOfSight : MonoBehaviour {

    ...

    private void InitializeLayerMask ()
    {
        layerMask = 1 << LayerMask.NameToLayer("Instrument");
    }
    ...

}
```



Capas en Unity, seleccionada la capa Instrument

Para dar un mayor realismo a este sentido, además, se ha añadido un factor de retardo, *timeBetweenUpdates***.** El NPC no será capaz de ver al instrumento automáticamente cuando este entre en su campo de visión, si no que tendrá que esperar por el tiempo de reacción del NPC. También ha sido parametrizada la distancia a la que es capaz de ver nuestro NPC (*sightDistance*). Modificando estos dos parámetros, podemos cambiar en gran medida la capacidad de visión de nuestro NPC, y con ello, su habilidad en el juego.

```
public class LineOfSight : MonoBehaviour {

    ...

    public float sightDistance = 1;
    public float timeBetweenUpdates = 0.2f;

    ...

}
```

Durante la ejecución del juego, se puede observar el rayo dibujado en pantalla como línea de depuración. Existe también en la GUI un botón para conectar / desconectar esta línea. Los atributos sightDistance y timeBetweenUpdates son públicos para que puedan ser modificados desde el inspector de Unity3D.

3.4.2.2. Representación del mundo.

El NPC necesita saber, para una correcta toma de decisiones, en qué estado se encuentra el mundo a su alrededor. En concreto necesita saber qué instrumentos hay ya colocados en cada pentagrama y a qué posición debería moverse, una vez tomada la decisión correspondiente. Para conseguirlo, se utilizará una alternativa usada ampliamente en inteligencia artificial para videojuegos: el NPC hará **trampas**. Nuestro NPC podrá acceder de forma privilegiada a información que, a priori, no es capaz de obtener.

En esta ocasión, nuestro NPC no utilizará su sentido de la vista para ver qué instrumentos están colocados. Este sentido sería demasiado complejo y lento, y no aportaría verdadero realismo al NPC. En su lugar, el NPC guarda una referencia a los pentagramas, de forma que puede acceder en todo momento a su contenido y a su posición. Esta implementación se encuentra en la clase WorldRepresentation.

Gracias a este “sentido”, el NPC podrá tomar decisiones correctas en función de los instrumentos ya colocados, y, desplazarse hacia la posición correspondiente a la hora de colocar un nuevo instrumento.

3.4.3 Módulo de Toma de Decisiones

El módulo de toma de decisiones se ha implementado utilizando dos técnicas distintas. Por una parte, se han implementado máquinas de estados, por ser la técnica más ampliamente utilizada en la mayoría de videojuegos y, por otra, árboles de comportamiento, por ser la técnica que más popularidad está ganando en los últimos años. Como referencia, se ha utilizado el libro *Unity 4.x Game AI Programming* [5].

Atendiendo a la forma de jugar de los distintos jugadores a los que se dio a probar el juego, se observó que existían dos personalidades predominantes entre los jugadores. La mayoría de los jugadores intentaban ganar la partida colocando correctamente los instrumentos en su pentagrama, pero algunos jugadores, por el contrario, disfrutaban impidiendo a su rival ganar, sin intentar ser ellos los ganadores. Para ello, se dedicaban a cambiar los instrumentos de su rival, no permitiéndole en ningún momento completar su partitura. Nuestro módulo de toma de decisiones incluirá ambos comportamientos como dos estrategias posibles: **ganadora** y **no perdedora**.

3.4.3.1. Máquinas de estado.

La implementación de las máquinas de estados que hemos realizado sigue el siguiente diagrama de clases:

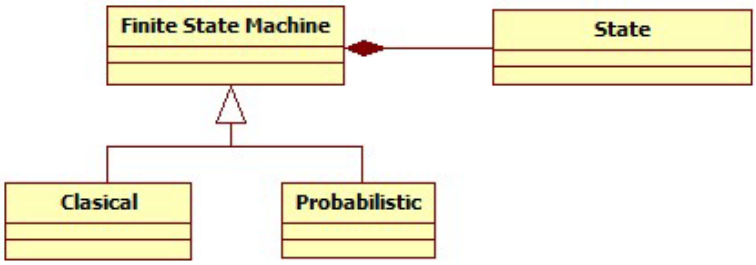


Diagrama de Clases para las Máquinas de Estados

La clase abstracta *Finite State Machine* es el núcleo de la máquina de estados. Contiene los distintos estados, y es el encargado de realizar las actualizaciones pertinentes en cada frame. En cada actualización, se comprueba si se debe cambiar de estado, y a continuación, se actualiza el estado actual.

```
public abstract class FiniteStateMachine<T> : MonoBehaviour {
    ...
    void Update()
    {
        if (ShouldChangeState())
            SwapStates();

        currentState.UpdateState();
    }
    ...
}
```

En cada actualización de un estado, este debe comprobar sus transiciones y, en caso de cumplirse las condiciones necesarias para que se produzca una transición, el propio estado indica a la máquina que nuevo estado debe ser el siguiente en ejecutarse. A continuación, el estado realizará el comportamiento que le está asociado. Por tanto, en nuestra implementación, los distintos estados están estrechamente ligados tanto al sistema de percepción como al de reacción.

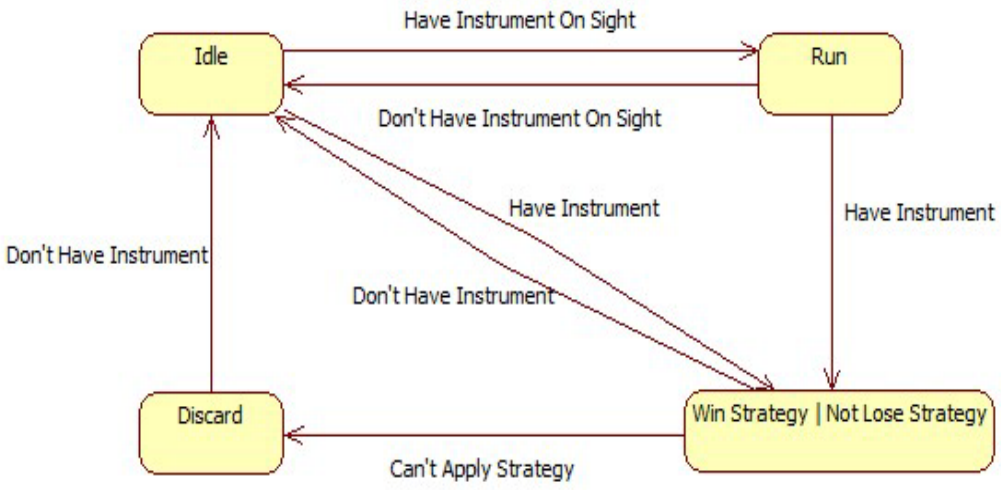
```
public abstract class State<T> {
    ...
    public void UpdateState()
    {
        DoBehaviour();
        CheckTransitions();
    }
    ...
}
```

Por tanto, las transiciones se encuentran dentro del propio estado. Este detalle tiene dos inconvenientes principales. El primero de ellos es que los diseñadores del juego necesitan

tener conocimientos de programación para poder modificar las transiciones, o solicitar estos cambios a los programadores. Por otra parte, si un comportamiento va a ser usado por dos NPC's con transiciones distintas, aunque el comportamiento sea el mismo, se deberá duplicar el estado para las nuevas transiciones. A continuación muestro un fragmento de código que representa las transiciones del estado "Idle", que es el estado de espera. El atributo playerController es la referencia al sistema de percepción.

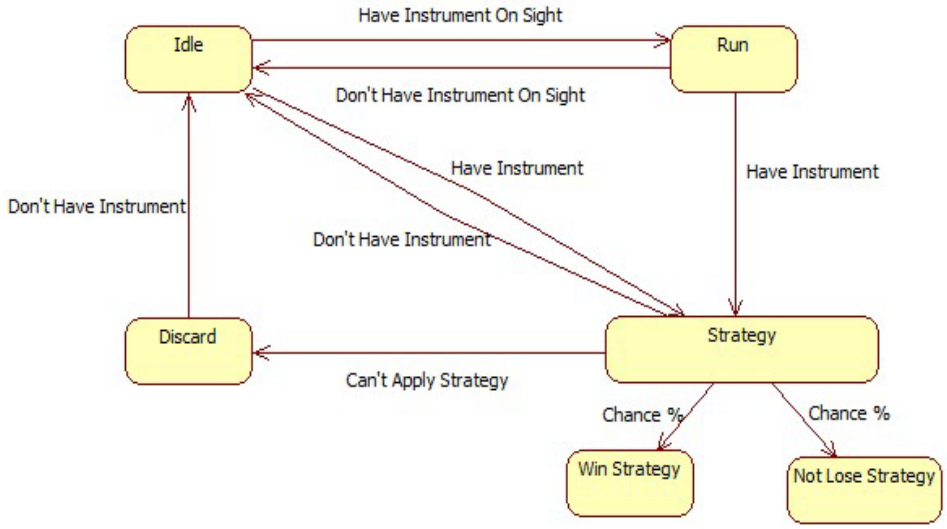
```
public class IdleState : State<GameObject> {
    ...
    public override void CheckTransitions()
    {
        if (playerController.PlayerHasInstrument())
            this.fsm.ChangeState("Strategy");
        if (playerController.HasInstrumentOnSight())
            this.fsm.ChangeState("Run");
    }
    ...
}
```

Siguiendo estos pasos, en Beat Fighters se han implementado las dos estrategias distintas como dos máquinas de estados clásicas, siguiendo el siguiente diagrama.



Máquina de Estado básica.

Para poder dar al NPC la posibilidad de elegir entre las dos posibles estrategias, se implementaron también las máquinas de estados finitas **probabilísticas**. Este tipo de máquinas se utilizan para poder tener comportamientos distintos para un mismo estado, de forma que se reduzca la predictibilidad de la máquina. Para ello, a cada posible comportamiento en un mismo estado se le aplica un peso, y a partir de este peso se realizaría la selección. La máquina de estado queda de la siguiente manera.



Máquina de Estado probabilística.

Para modificar las probabilidades dentro de Beat Fighters y ver cómo cambia el comportamiento asociado, es suficiente con modificar las probabilidades asignadas en el siguiente fragmento de código.

```
public class ProbabilisticFSM : ProbabilisticFiniteStateMachine<GameObject>
{
    void Start () {
        ...
        this.AddState("Strategy", new WinStrategyState(this.gameObject), 50);
        this.AddState("Strategy", new NotLoseStrategyState(this.gameObject), 50);
        ...
    }
}
```

En Beat Fighters, durante la ejecución demostrativa de las máquinas de estados finitas, también es posible cambiar de máquina en vivo, pudiendo observarse los distintos comportamientos posibles. Se muestra también los estados de los que se compone la máquina, y en caso de las probabilísticas, su probabilidad, así como qué estado se encuentra activo en cada momento.

3.4.3.2. Árboles de comportamiento.

Desde el lanzamiento de Halo 2 en 2004, por Bungie Studios el uso de árboles de comportamiento ha ganado una popularidad abrumadora frente a las máquinas de estados clásicas. Sus principales ventajas frente a las máquinas de estados son su alta capacidad de reutilización de tareas y su accesibilidad para los diseñadores.

En la implementación de las máquinas de estados, encontramos que, como el propio estado controla sus transiciones, empieza a ser necesario ciertos conocimientos de programación para poder gestionarlas, conocimientos que no necesariamente los diseñadores deben tener. Además, también nos impide llevarnos el comportamiento del estado a otra máquina que buscase el mismo comportamiento pero con otras transiciones pues nos obligaría a reescribir el estado.

Los árboles de comportamiento, con su filosofía de tareas atómicas y la eliminación de las transiciones explicitas buscan simplificar este procedimiento.

La implementación realizada sigue el siguiente diagrama de clases:

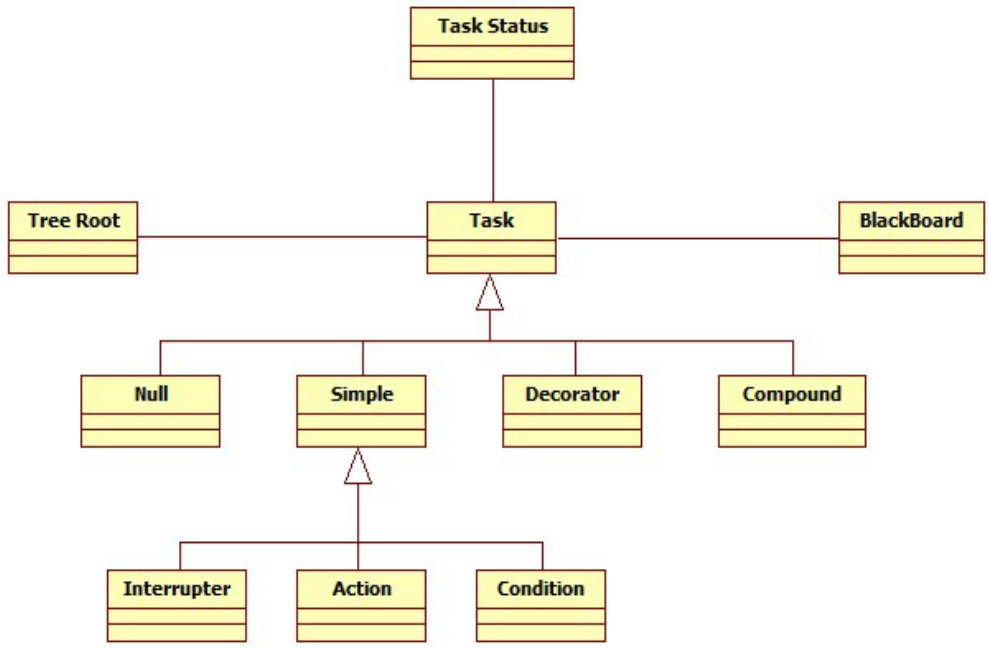


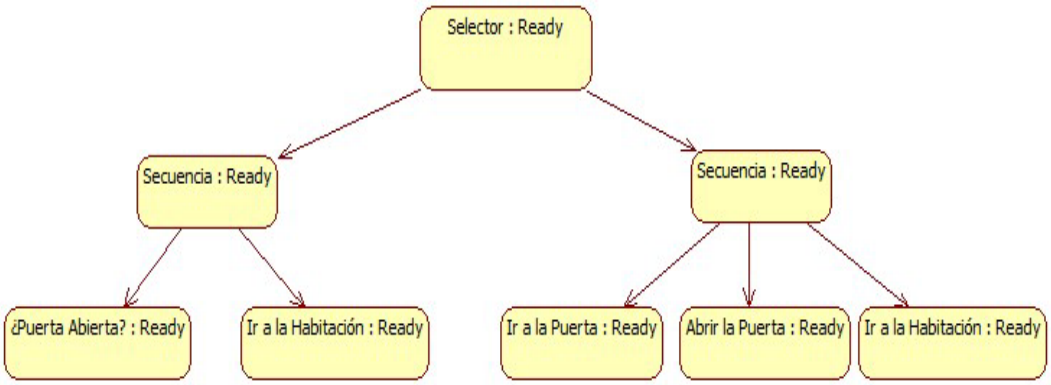
Diagrama de clases para los Árboles de Comportamiento.

Como se puede observar, el corazón del árbol está en las tareas. Las tareas representan el comportamiento a realizar, y deben ser lo más atómicas posibles. Aun así, es posible que cada tarea necesite más de un frame para acabar su ejecución. Para controlar esta situación, a cada tarea se le asocia un **estado de ejecución (*Task Status)***. Los posibles estados en que puede estar una tarea son:

- Listo (Ready).
- En ejecución (Running)
- Terminado con éxito (Success).
- Terminado en fracaso (Failed).
- Interrumpido (Interrupted).

En cada actualización del árbol, se busca la primera tarea que esté lista para ser ejecutada, o bien, que ya esté en ejecución. Ilustraremos este funcionamiento con un ejemplo:

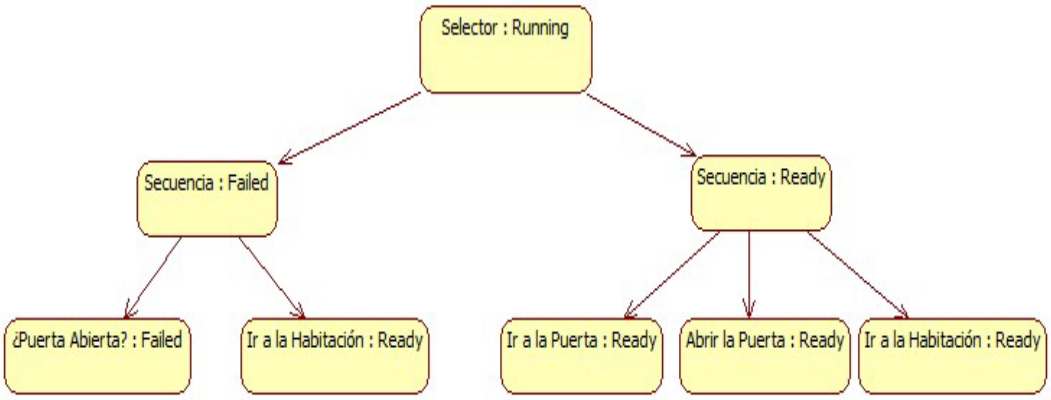
Nuestro NPC está intentando ir de una habitación a otra. Para ello, debe comprobar si la puerta está abierta, en caso afirmativo, simplemente atravesarla. En caso de que esté cerrada, deberá abrirla, y entonces, atravesarla. Al principio, el árbol tendrá la siguiente forma:



Ejemplo de Árbol de Comporamiento: Estado inicia

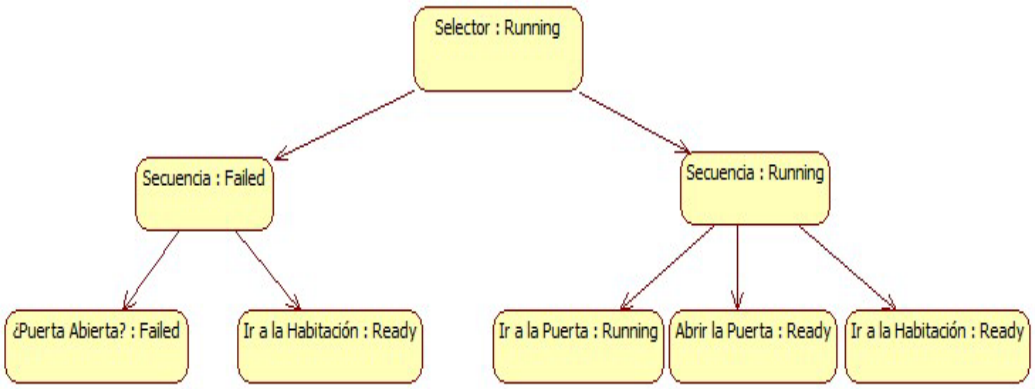
En el primer frame, empezará la ejecución de la primera tarea. Esta es el selector que hace de raíz del árbol. La tarea selectora probará a sus hijos en orden hasta que uno de ellos consiga acabar con éxito. Por tanto, en este frame, esta tarea probará a ejecutar, de sus subtareas, la primera que no haya fallado aún. Encuentra de esta manera, la primera de las tareas secuencia, que también se encuentra lista para ser ejecutada. Esta tarea, a su vez, intentará ejecutar a todas sus hijas en orden, acabando con éxito si todas sus subtareas lo consiguen. La primera de sus tareas, "¿Puerta Abierta?" se encuentra lista para ser ejecutada, por lo que será ejecutada en este frame.

Tras el primer frame, resulta que la puerta está cerrada, por lo que el estado "¿Puerta abierta?" falla, así como la secuencia a la que esta condición pertenece. Sin embargo, la tarea selectora permanecerá en ejecución, pues probará con la siguiente alternativa. El árbol queda así:



Ejemplo de Árbol de Comporamiento: Falla "¿Puerta Abierta"

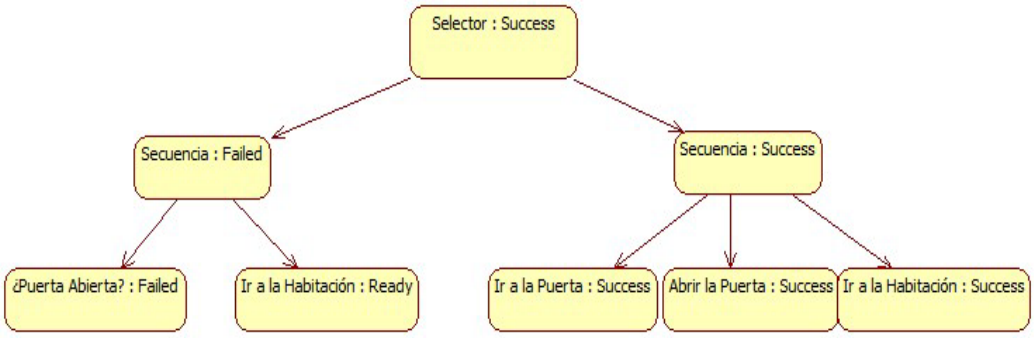
Ahora, el NPC debe abrir la puerta. Para ello, en cada frame realizará cada una de las acciones de la segunda secuencia, mientras que ninguna falle.



Ejemplo de Árbol de Comporamiento: Se dirige a la puerta

Probablemente la tarea "Ir a la Puerta" sea una tarea que lleve más de un frame. Por tanto, esta tarea se mantendrá en estado de ejecución hasta completarse. En cada frame, el árbol ejecutará siempre primero las tareas que ya se encuentren en ejecución.

Finalmente, el NPC ha sido capaz de acabar la secuencia con éxito, por lo que el selector también finaliza con éxito, y se deduce que el comportamiento ha sido completado.



Ejemplo de Árbol de Comporamiento: Consigue abrir la puerta y atravesarla

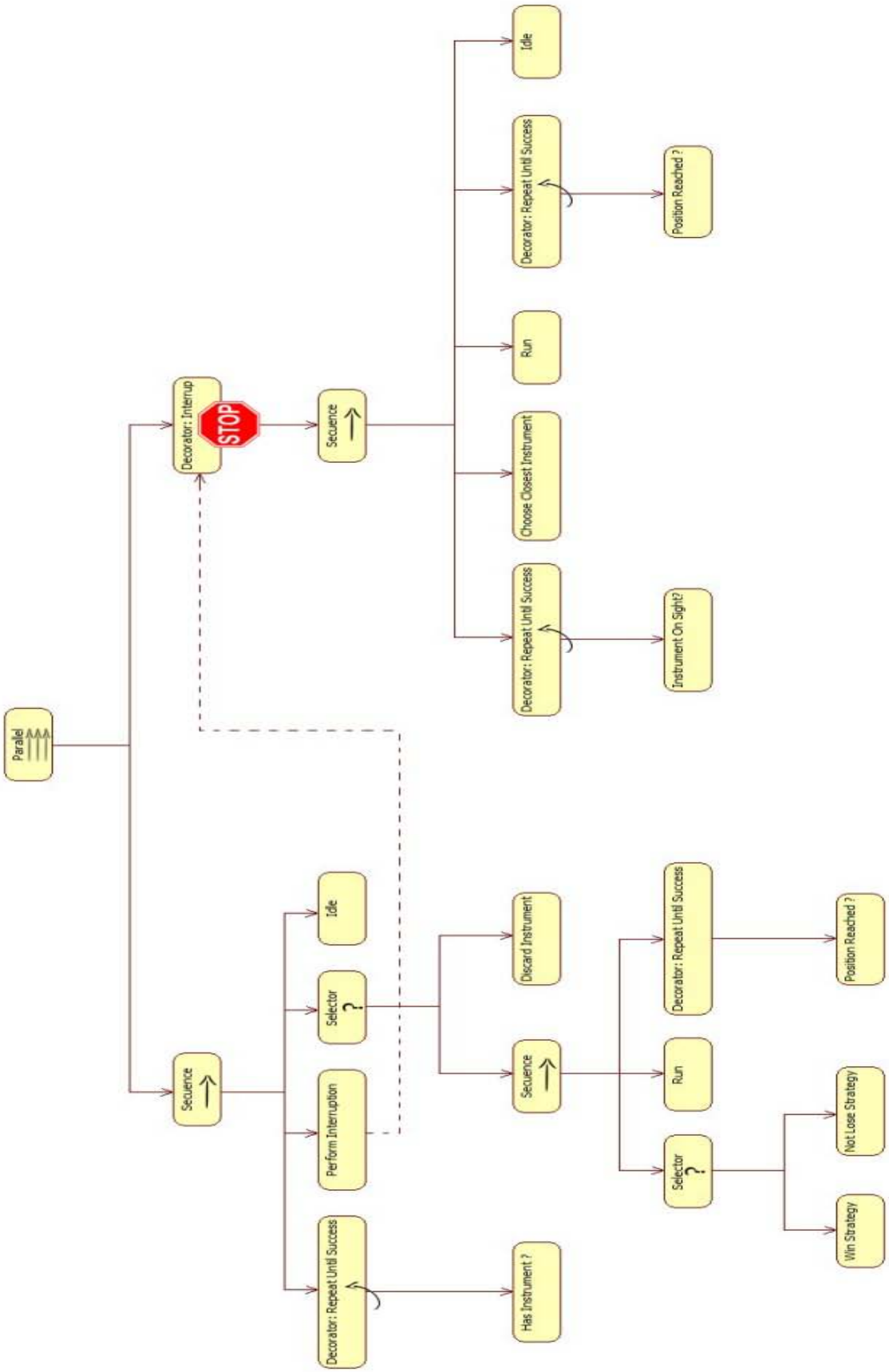
Algunos estados necesitarán comunicarse entre sí para transmitirse datos calculados que son necesarios más adelante. Para ello, se crean las **pizarras (*blackboards*).** Al crear una nueva tarea, se le asigna una pizarra. Se pueden tener tantas pizarras como se desee, de forma que no haya colisiones entre ellas. Las tareas podrán tanto escribir como leer de ellas los datos que necesiten. En nuestra implementación, la responsabilidad de saber qué tipo de dato se debe leer / escribir recae, por simplicidad, en las propias tareas.

En nuestra implementación, se distinguen entre tareas simples (no tienen hijos), decoradoras (tienen un hijo), y compuestas (pueden tener multitud de hijos). Además, y por motivos didácticos, se mantiene la diferencia entre tareas condicionales y de acción dentro de las tareas simples. Ambos tipos de tareas comparten la misma interfaz, pero el comportamiento esperado de cada una de ellas es distinto. Por una parte, las condiciones consultarán alguna condición del mundo, mientras que las acciones realizarán cambios en el mismo.

Los decoradores implementados siguen la filosofía del patrón de diseño **Decorador**. Este patrón busca otorgar de forma dinámica cierta funcionalidad a un objeto concreto, sin necesidad de crear nuevas clases que hereden del objeto principal y añadan este comportamiento. Por poner un ejemplo, podríamos querer repetir una cierta tarea hasta que esta tenga éxito. Este comportamiento (repetirse hasta tener éxito) no pertenece a la tarea en concreto, pero en un momento dado, podríamos permitirle esta capacidad mediante el decorador adecuado.

Finalmente, para dar la capacidad de poder interrumpir un subárbol a otro subárbol en cualquier momento, se ha utilizado una tarea decoradora emparejada con una tarea simple. El decorador dará al subárbol decorado la capacidad de ser detenido y estará asociado a una (o más) tareas simples interruptoras. Estas tareas, al ejecutarse, detendrán al árbol indicado por la tarea decoradora que tengan asociada.

En Beat Fighters, el árbol de comportamiento que utiliza el NPC es el siguiente:



Árbol de Comporamiento implementado en Beat Fighters

En Beat Fighters, durante la ejecución demostrativa de los árboles de comportamiento, se puede observar el estado en todo momento de las distintas tareas, así como el de las dos pizarras (una por cada subárbol de la raíz). Al igual que en los árboles de comportamiento, es posible ver la línea de visión del NPC, así como desactivarla. Para una mayor comprensión de lo que está pasando se recomienda utilizar el modo de ***Slow Motion*** durante la ejecución de este módulo, pues las tareas cambian a gran velocidad.

3.5. Conclusiones

El desarrollo del apartado de inteligencia artificial de este proyecto permite la extracción de bastantes aprendizajes. Por una parte, nos encontramos ante una disciplina compleja y madura que engloba muchísimas disciplinas en su interior, así como una gran cantidad de soluciones para todos los posibles problemas que puedan aparecer.

Por una parte, la visión de la inteligencia artificial como un sistema se ajusta a la perfección a la sensación buscada en el desarrollo de videojuegos. Este perfecto ajuste permite guiar el desarrollo, descubriéndonos que no es importante únicamente el módulo de toma de decisiones, si no que el sistema de percepción es también muy importante. De nada nos sirve tener un NPC capaz de tomar unas decisiones correctas en el 100% de las situaciones si, sin embargo, es completamente ciego al mundo que le rodea.

En cuanto al desarrollo de Beat Fighters, nos ha permitido aprender lo importante de la conexión entre el sistema de percepción y el de decisión. Por su parte, hemos aprendido para el sistema de percepción una técnica tan válida como cualquier otra puede ser hacer trampas. Lo importante es que el comportamiento emergente parezca realista, mientras que el jugador no tenga la sensación de que la IA realmente está tomando ventaja a escondidas.

En el módulo de toma de decisiones vivimos la evolución de un sistema clásico, las máquinas de estado, en un sistema algo más refinado y complejo para la obtención de, aparentemente, los mismos resultados. Aunque las máquinas de estado siguen siendo una técnica utilizada y perfectamente válida para el desarrollo de inteligencias sin demasiada complejidad, se vuelven tediosas de utilizar cuando las necesidades del proyecto crecen. Su gran inconveniente, como ya se ha insistido durante el presente documento, así como la gran ventaja de los árboles de decisiones, es su dificultad de manejo para los diseñadores. El desarrollo de videojuegos compete a distintos perfiles de personas y esto se ve reflejado en situaciones como esta, en que existe una gran preocupación por hacer accesible ciertos aspectos de desarrollo a los distintos roles.

Finalmente, y como posible mejora para el futuro, sería muy interesante desarrollar una herramienta que permitiera, de manera visual, construir los árboles de comportamiento a los diseñadores. De esta manera, conseguiríamos aprovechar por completo la potencialidad de los mismos.

Capítulo 4: Redes

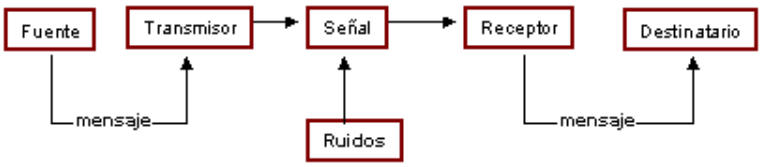
"En internet nadie sabe que eres un perro"
Peter Steiner, chiste en The New Yorker, Julio de 1993

4.1. Introducción

El siglo XXI ha sido bautizado, por méritos propios como la era de la información. Sin duda, vivimos años de importantes avances tecnológicos y profundos cambios sociales. Los avances en redes informáticas han permitido a la sociedad superar barreras físicas que hasta hace bien poco resultaban casi infranqueables. No solo está el hecho de que cualquier acontecimiento importante que suceda recorra el globo terráqueo en cuestión de segundos para que cualquier gobierno, empresa o agente pueda reaccionar en consecuencia. Es que, cada persona, cada individuo, aporta y consume información a raudales casi a cada paso que da.

El concepto de red se encuentra presente en el día a día de todos nosotros. Nadie concibe ya la vida sin internet. La tecnología móvil, las redes 3G y 4G están a la orden del día, permitiendo que cualquiera pueda acceder desde su Smartphone a toda la información del mundo, y sobre todo, a la de sus semejantes. Ha nacido el concepto de redes sociales, de la mano en gran parte de Facebook, y, con esto, nuevas formas de comunicación. El correo ordinario ha sido desplazado a un segundo plano con la irrupción de los servicios de correo electrónico, mientras que el teléfono se utiliza más para conversaciones a través de aplicaciones como WhatsApp que para llamar.

Siguiendo esta línea de pensamiento, podemos decir que, sin duda, los avances en redes informáticas han traído consigo nuevas formas de comunicación. Según la RAE, comunicar (del Latín *commūnicāre,* que significa "*compartir*") estaría definido como la *"Transmisión de señales mediante un código común al emisor y al receptor".* De esta definición se puede extraer que, para que la comunicación exista hacen falta al menos dos interlocutores (no necesariamente personas). Un emisor, interesado en transmitir un mensaje, y un receptor, interesado en recibirlo. El mensaje debe estar codificado de manera que ambos interlocutores puedan entenderlo. Finalmente, el mensaje debe transmitirse a través de un canal en el que los interlocutores sean sus extremos *(Modelo de Shannon y Weaver)*.



Modelo de Comunicación de Shannon y Weaver

Las tecnologías de red actuales podrían analizarse desde dos puntos de vista. Para la comunicación humana o de los elementos en los extremos de la red, analizaríamos la tecnología como el canal para poder compartir la información deseada. Pero, si analizamos la propia tecnología, descubrimos que, en sí misma, cumple punto por punto todos los puntos definidos previamente, estableciendo conexiones entre varios puntos, utilizando códigos para la transmisión de los mensajes…

La gran ventaja que han aportado las nuevas tecnologías ha sido la casi total destrucción de las barreras físicas que antes evitaban la comunicación. Actualmente una persona en un punto cualquiera del mundo puede comunicarse con otro individuo alejado por cientos o miles de kilómetros. Y, por supuesto, también pueden **jugar**.

4.2. Otro poco de Historia

El artículo *Brief History of the Internet*[8] realiza un interesante recorrido por la historia de Internet, contado por sus protagonistas directos, que pasamos a resumir a continuación.

La primera descripción de las interacciones sociales que podían hacerse posibles a través del concepto de red se encuentran en una serie de artículos escritos por el trabajador del Instituto Tecnológico de Massachusetts (MIT) J.C.R. Licklider en 1962. En estos artículos desarrolla el concepto de "Red Galáctica". Según Licklider, esta red permitiría a ordenadores en distintos lugares del mundo conectados entre sí acceder a datos y programas almacenados en cualquier otro equipo de la red. En esencia, este concepto es muy similar al actual Internet. Licklider participó en varios programas de investigación dentro de DARPA (Agencia de Proyectos de Investigación Avanzados de Defensa), donde convenció a sus sucesores Ivan Sutherland, Bob Taylor y Lawrence G. Roberts de la importancia del concepto de interconexión.

Por otra parte, y también en el MIT, Leonard Kleinrock publicaba en 1961 el primer artículo sobre intercambio de paquetes, concepto que ampliaría con un libro en 1964. Kleinrock aseguraba que el intercambio de paquetes era una mejor forma de comunicación para los equipos informáticos que el uso de circuitos específicos. Para demostrarlo, desarrolló un sistema de red para intercambiar paquetes de información entre distintos equipos. Curiosamente, Paul Baran en la Corporación RAND y Donald Davies en Londres desarrollaban sistemas similares en paralelo.

Basándose en estos estudios, en 1965, Thomas Marill y Lawrence G. Roberts crean la primera red WAN, que conectó el TX-2 de Massachusetts con el Q-32 en California. Se comprobó que los equipos conectados en la red podían trabajar bien en conjunto, y que los circuitos telefónicos no eran la mejor solución para realizar este trabajo. Se confirmaban así las suposiciones de Kleinrock en lo que a la comunicación mediante paquetes se refiere. Se inician las investigaciones para el desarrollo de ARPANET dentro de DARPA, contratando a Lawrence G. Roberts como líder del proyecto.

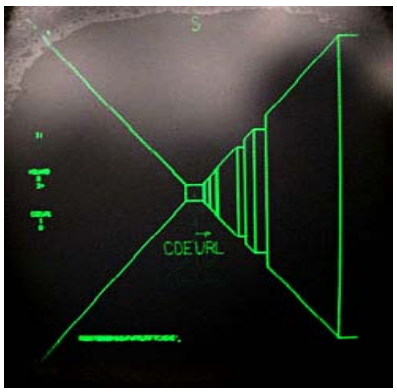
En septiembre de 1969 se conecta el laboratorio de Kleinrock en California, primer nodo de la red ARPANET, con el Instituto de Investigación de Stanford. Un mes más tarde, el primer mensaje es enviado a través de la red usando ya la tecnología de paquetes con éxito. Se añaden dos nodos más a la red, en las universidades de Santa Barbara y Utah. Ya en 1972, en la International Computer Communication Conference (ICCC), organizada por Bob Kahn, se realizó una demostración de las cualidades de ARPANET, cosechando un importante éxito. Fue en este año que se desarrolló la primera aplicación importante de la nueva red: el correo electrónico.

Primeros cuatro nodos de ARPANET

En este punto, ya estaban los cimientos de la red claramente asentados. A partir de entonces, empiezan a desarrollarse nuevos protocolos y aplicaciones para la red, hasta llegar a las potentes redes de nuestros días. Entre las aplicaciones que empezaron a aprovechar las capacidades de red están, por supuesto, los videojuegos.

Durante las décadas de los 70 y principios de los 80, la conexión a la red ya se ha popularizado. Aparece el sistema de *Tablón de Anuncios*. Esta aplicación permitía a los usuarios conectados al sistema intercambiar "anuncios". Datos, mensajes, correos… básicamente, texto. Fue en estos sistemas donde surgieron los primeros atisbos de juego online. Los juegos de rol, que también se popularizaron en estos años, eran jugados a través de estos sistemas, tratando de traspasar la experiencia que los jugadores sentían en la mesa a la pantalla.

La primera evolución hacia juegos con interfaz gráfica en la red se produjo con la creación del llamado ANSI-Art, que, basado en caracteres ANSI, permitía crear interfaces pseudo-gráficas, y que fue utilizado para el desarrollo de juegos del estilo de los juegos de casino. En 1974 vería la luz "*Maze War*", el primer videojuego online propiamente dicho. Consistía en un laberinto en el que los jugadores se desplazaban buscando a sus contrincantes para disparar contra ellos. Los otros jugadores estaban representados en el juego como ojos. En este mismo año, el juego "*Spasim*" marca un nuevo hito al convertirse en el primer juego online en 3D. En este juego, que podía soportar hasta 32 jugadores simultáneos, los jugadores podían explorar una galaxia compuesta por 4 sistemas planetarios, mientras veían a los demás jugadores moverse también representados por unas naves espaciales formadas por una malla.



Maze War, aunque no lo parezca, precursos de los juegos de disparos.

El siguiente hito lo marcaría "*DOOM*" de Id Software, ya en 1993. Este juego, considerado como uno de los juegos más influyentes de la historia incluyó por vez primera un modo multijugador online, permitiendo a los jugadores disfrutar de una experiencia nunca antes vista. No tardaría Blizzard en darse cuenta del potencial del juego online, ofreciendo experiencias orientadas totalmente hacía el modo multijugador con sus sagas "*Diablo*" (1996) y "*Starcraft*" (1998), siendo este último aún jugado en el territorio de Corea del Sur.

Los videojuegos online siguieron haciéndose populares en los años siguientes. En 1998, Valve lanza *"Half Life"*, juego de disparos en primera persona orientado a la experiencia mono jugador, pero con características multijugador. Sin embargo, utilizando el mismo motor, lanza distintos mods, incluyendo juegos como *"Counter Strike", "Team Fortress" y "Day of Defeat"*. Estos juegos se convirtieron en clásicos automáticos, colmando los ciber cafés de jugadores. Partidas de "*Counter Strike"* y de *"Starcraft"* eran retrasmitidas por televisión, sentando las bases de los actuales e-sports.

Vista aérea del torneo SKY ProLeague 2005 de Starcraft

Aparecen nuevos géneros para incluir a los nuevos juegos online. Especialmente popular es el género MMORPG, en el que los jugadores se mueven por un mundo abierto y persistente, compartiendo aventuras con jugadores de todo el mundo. *"World of Warcraft"* (2005)*, "Guild Wars"* (2005) y *"Lineage 2"* (2003) se convierten en los referentes del género. En la actualidad, se impone el género MOBA, siendo el género más rentable, aún siendo juegos gratuitos financiados a través de ventas en el juego. *"League of Legends"* es actualmente el juego con la mayor comunidad de jugadores.

4.3. Redes en Videojuegos

En la actualidad, los videojuegos orientados a la red están a la orden del día, con géneros propios como el MMORPG. Cualquier juego AAA que se precie debe tener al menos una parte online, o al menos social, quedando la experiencia de juego en solitario clásica reservada para juegos de menor calibre o para la esfera indie. Sin embargo, la decisión de desarrollar un videojuego con conceptos de red no debe ser tomada a la ligera.

Las características de red de un videojuego deben estar definidas ya en la fase de diseño del mismo. De esta manera, se podrá garantizar una experiencia interesante para los jugadores involucrados, así como un correcto análisis y definición de los requisitos a la hora de implementar el juego. Recordemos que, si ya de por si en videojuegos el rendimiento es crítico, durante la experiencia online se vuelve aún más importante.

Durante la etapa de diseño hay que tener en cuenta multitud de aspectos. En primer lugar, hay que plantear el **papel de la red** con respecto al resto del juego. Obviamente no se desarrollan igual un juego como el *"World of Warcraft*", que no tienen sentido sin la red, como una aventura gráfica como podría ser *"The Secret of Monkey Island*", que, directamente, no tiene ninguna característica de red. El peso que la parte de red tiene dentro del diseño de juego influye drásticamente en la tecnología y arquitecturas que serán utilizadas en la fase de desarrollo. Por supuesto, hay juegos que jamás podrían haber existido de no ser por los avances tecnológicos de los últimos años.

Dentro de la etapa de diseño, debemos especificar si el juego está orientado a conexión, es decir, si es en **tiempo real**. Dentro de los juegos que no son en tiempo real tendríamos, por ejemplo, el popular *"Apalabradros"*. En este juego, que es una revisión del clásico Scrabble de mesa, después de que un jugador realice un movimiento se notifica a su rival, que puede realizar su movimiento cuando lo desee. Sin embargo, en "*FIFA World"*, juego de fútbol online, ambos jugadores ven los movimientos del rival en el campo instantáneamente, siendo este aspecto crítico para el mismo.

Otro aspecto a tener en cuenta en fase de diseño es la **persistencia**. ¿Existe alguna información que debe conservarse aun cuando el jugador abandone la partida? En el ya mencionado *"FIFA World"*, no necesitamos conservar ninguna información sobre el partido una vez el jugador se desconecta, pero, sin embargo, si se conserva cierta información sobre el resultado del mismo, con el fin de rellenar mecánicas que no son puramente online, así como completar el perfil del jugador. Por otra parte, en "*World of Warcraft"* el mundo es permanente en el servidor. Si un grupo de jugadores ya ha eliminado a cierto enemigo, otros no lo encontrarán hasta que se regenere.

Estos factores son los más importantes a la hora de diseñar un juego que va a contener parte online. Por supuesto, hay más aspectos, como puede ser la plataforma a la que va dirigido (¿Será un pc con conexión permanente a internet a más de 10mbps, o un Smartphone con una pequeña tarifa de datos?), el territorio al que va dirigido (Los 6,6mbps de media españoles no pueden competir contra los 21,9mbps de Corea del Sur), el tipo de jugador al que nuestro juego está dirigido…

Todos estos aspectos de diseño, como ya se ha comentado, influirán de una manera u otra, tanto en la tecnología a usar, como el modelo y la arquitectura. A continuación, paso a analizar las principales tecnologías de red existentes, así como distintos modelos y arquitecturas para el desarrollo de videojuegos en red.

4.3.1. Tecnologías

Tres son las principales tecnologías utilizadas a la hora de desarrollar videojuegos en red. Estas son:

1. Redes Dial-Up y PPP (Protocolo punto a punto)
2. Redes TCP/IP.
3. Redes Wireless.

Las redes dial-up actualmente ya no se usan para el desarrollo de videojuegos, pero permitieron los primeros juegos en red, las redes Wireless han sido utilizadas sobretodo en el mundo portátil, pero, en su mayor parte, la tecnología TCP/IP se ha impuesto como la más ampliamente utilizada. A continuación describiremos brevemente cada una, basándonos en los apuntes utilizados en el Máster de Desarrollo de Videojuegos de la UCM[10].

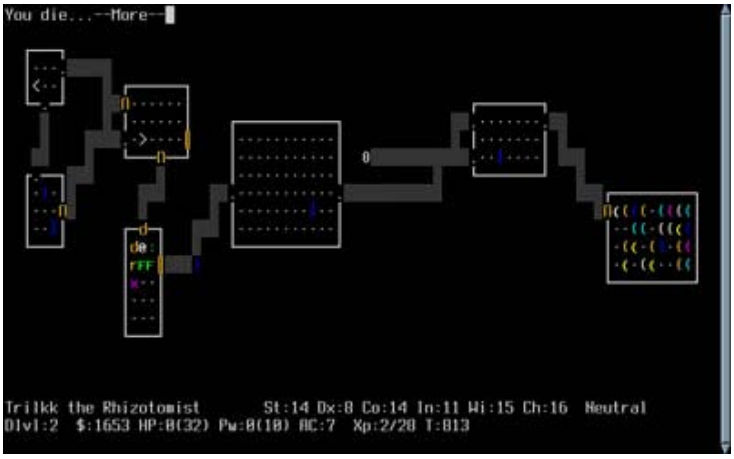
4.3.1.1 Redes Dial-Up

Redes Dial-Up

Las redes Dial-Up o redes de conexión por línea conmutada son una forma de conexión en la que cada cliente utiliza un módem con el que realiza una llamada a través de la línea telefónica a su nodo ISP (Internet Service Provider). Se trata de un tipo de conexión lenta, pero es factible en la mayor parte del planeta. Esto es debido a que la línea telefónica se encuentra extendida en prácticamente todo el mundo, mientras que la infraestructura necesaria para las conexiones de banda ancha está, en comparación, mucho más limitada.

Se apoya en el protocolo PPP, o punto a punto. Este protocolo se utiliza para establecer una conexión directa entre dos nodos de la red. Entre los servicios que proporciona, se encuentran la autenticación de conexión, el cifrado de la transmisión y compresión de datos.

En los albores de internet, se utilizó este tipo de conexión para desarrollar los primeros juegos online. Se trataba de juegos de Dragones y Mazmorras que, para la comunicación entre los jugadores, hacían uso de un **Sistema de Tablón de Anuncios** (Bulletin Board System). Este software permite a los usuarios conectarse al sistema y, entre otras cosas, intercambiar mensajes con otros usuarios. Son, de alguna manera, los precursores de los foros de internet. Los primeros juegos (que se jugaban en modo texto) utilizaban este sistema para transferir la experiencia de juego de una partida de mesa a una experiencia a larga distancia.

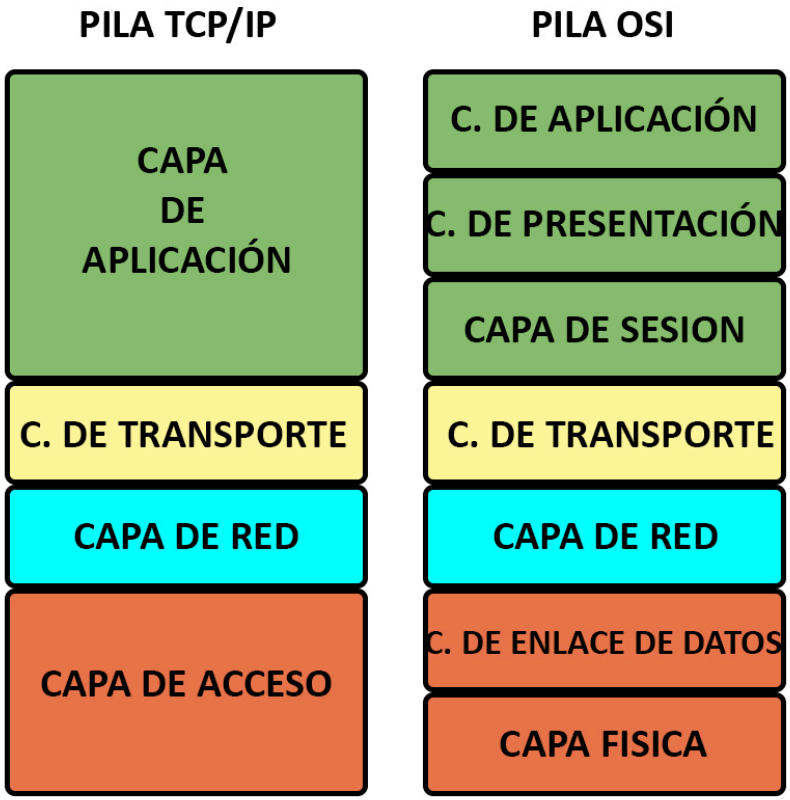


Juego de Dragones y Mazmorras en un tablón de anuncios

4.3.1.2 Redes TCP/IP

El modelo TCP/IP es, en realidad una descripción de protocolos de red que fue implantado en la red ARPANET, predecesora de Internet. Describe una guías generales de diseño e implementación de protocoles de red específicos para permitir que un equipo pueda conectarse a una red. Especifica como los datos deben ser formateados, direccionados, transmitidos, enrutados y recibidos por el destinatario.

El modelo TCP/IP es un modelo divido en capas, al igual que le modelo OSI, pero frente a las siete capaz de este, TCP/IP cuenta solamente con cuatro capas. Estas capas son:



Pilas TCP/IP y OSI

Capa 1 o capa de acceso: Se corresponde con las dos primeras capas de OSI (Física y enlace de datos). En esta capa se encuentran los protocolos necesarios para manejar todo lo concerniente al manejo físico de los datos. Se encontrarían en esta capa protocolos como el ya citado PPP o Ethernet.

Capa 2 o capa de internet: Se corresponde con la tercera capa del modelo OSI (Capa de red). Es la encargada del enrutado de los paquetes a través de la red. El protocolo más representativo de esta capa es el protocolo IP.

Capa 3 o capa de transporte: Se corresponde directamente con la capa de transporte del modelo OSI. Esta capa se encarga del empaquetado de los datos, control de errores y de flujo, seguridad de la transmisión… Se trata de una de las capas más interesantes a la hora de desarrollar videojuegos, pues se debe decidir que protocolo se usará, eligiendo entre **UDP y TCP.**

El protocolo TCP es un protocolo orientado a conexión. Para ello, se produce un intercambio de paquetes inicial en el que se establece la conexión entre los equipos. Además, se caracteriza por su alta fiabilidad, pues se garantiza que los datos llegarán intactos y en el mismo orden en que fueron enviados. Puede, en caso de error, realizar procedimientos para recuperarse de los errores. Sus principales desventajas son el tamaño de los paquetes (20 bytes de cabecera) y que, para poder soportar todas sus capacidades de fiabilidad, se renuncia en parte a la velocidad, incluyendo tiempos de espera adicionales utilizados para el establecimiento de la conexión y el acuse de recibo de los datos. *World of Warcraft* es quizás el videojuego más famoso que utiliza este protocolo.

Por otra parte, el protocolo UDP no se orienta hacia la conexión. Esto le permite enviar datos a otro equipo sin esperar ningún tipo de respuesta por parte del destinatario. Aboga por la velocidad de la conexión frente a la fiabilidad de la información. Por tanto, no implementa ningún tipo de método que asegure que los paquetes lleguen ordenados, o, si quiera, que los paquetes lleguen a su destino. Esta responsabilidad es delegada a la capa de aplicación. Los paquetes UDP son más pequeños que los TCP (8 bytes de cabecera). *Final Fantasy XI,* entre otros, apostó por UDP.

Existe también RUDP, acrónimo para Reliable-UDP. Se trata de una versión de UDP que, en capa de aplicación, implementa gran parte de los mecanismos de TCP. En los últimos años se está convirtiendo en un protocolo muy popular en el desarrollo de videojuegos pues aporta las ventajas de fiabilidad de TCP, pero sin congestionar la red. Unity3D sólo nos permite trabajar con RUDP o UDP directamente.

Capa 4 o capa de aplicación: Aquí es donde se encuentran las aplicaciones propiamente dichas (en nuestro caso, los videojuegos) Se encarga de la representación y codificación de los datos. En el modelo OSI, se correspondería con las tres últimas capas (Sesión, Presentación y Aplicación).

4.3.1.3 Redes Wireless

Existen ocasiones en que los jugadores no disponen, en un momento dado, de acceso a Internet. Esto puede deberse a diferentes motivos, como que, simplemente, no exista la infraestructura necesaria en un determinado lugar. Este problema se vuelve especialmente sensible en las consolas portátiles, por su propia naturaleza de consolas orientadas al movimiento. La mítica Game Boy (que no disponía de ningún tipo de conexión a internet) utilizaba, para la conexión entre sistemas, el llamado "*Cable Link*". Este cable permitía conectar dos consolas entre sí para intercambiar datos o jugar juntos. La saga *"Pokemon"* explotó este recurso al máximo, lanzando diferentes versiones del mismo juego (por ejemplo, rojo y azul) con distinto contenido, incentivando el intercambio de dichas criaturas entre los usuarios.

Sin embargo, con la evolución de las consolas los sistemas mejoraron, aprovechando en todo momento la proximidad entre los jugadores. Siguiendo con el ejemplo de Nintendo, en su reciente Nintendo 3DS, se incluye un sistema de conexión entre los jugadores basado en la proximidad, sin necesidad de ningún tipo de infraestructura preexistente. Se trata del sistema "StreetPass". Esta funcionalidad de la consola le permite conectarse automáticamente a otras Nintendo3DS que se encuentren dentro de su radio de acción. Este sistema es utilizado por juegos como *"Super Street Fighter IV"*.

Nintendo siempre ha abogado por la conexión entre jugadores cercanos

Por su parte, Sony, implementó el sistema de redes AD-HOC en su consola PlayStation Portable (PSP). Se trata de un tipo de red inalámbrica descentralizada, en la que cada consola participa en el encaminamiento de datos mediante el reenvío de información hacia los otros nodos de la red. De esta manera, se crea una red en la que todos los participantes ocupan el mismo puesto. Esta tecnología fue uno de los puntos fuertes de la consola, siendo ampliamente explotada por juegos como la saga *"Monster Hunter"* de la nipona CAPCOM.

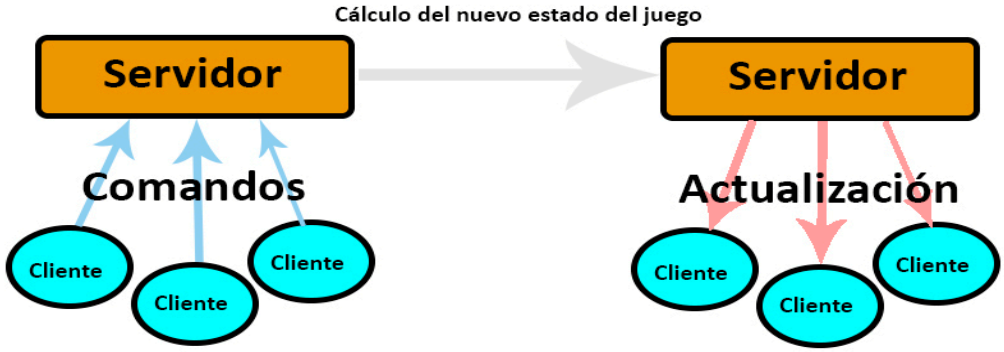
4.3.2 Modelos

Antes de empezar a definir diferentes arquitecturas para videojuegos en red, procederé a definir un par de términos importantes. El primero de ellos es el concepto de **Mundo Virtual.** Todos los jugadores conviven en un espacio común, que contiene tanto a los jugadores, como otros objetos, el propio escenario, las normas del juego… El estado del mundo virtual en un momento dado, es conocido como el **Estado del Juego.** Todos los jugadores deben tener la misma percepción del estado del juego en todo momento, y para ello, deben sincronizar su propio estado con el del resto de jugadores.

Por otra parte, el **modelo de juego** indica cómo se representa el mundo virtual y, sobretodo, como se gestiona, almacena y actualiza el estado del juego para cada jugador. La **arquitectura de red**, por su parte, define cómo se realizarán los intercambios de información entre los distintos jugadores. A continuación analizaremos los dos principales modelos de red utilizados en desarrollo de videojuegos, **cliente-servidor** y **p2p**.

4.3.2.1 Modelo Cliente – Servidor

Este modelo diferencia entre dos tipos de actores dentro del escenario. Por una parte están los clientes, que representan a los jugadores, y por la otra, el servidor único, que almacena el estado del juego. Esta es la característica principal de este modelo.



Modelo Cliente - Servidor

Durante una partida con servidor autoritario, cada jugador se conecta únicamente con el servidor, y es este el que toma la responsabilidad de hacer cumplir las normas del juego a todos los jugadores. Para ello, recibe de los clientes información sobre las acciones realizadas por los clientes, calcula el resultado de sus acciones e informa a todos los jugadores de lo sucedido, para que de esta forma puedan actualizar sus correspondientes estados de juego. Podemos extraer por tanto, que la comunicación es bidireccional y que solo se realiza de los clientes hacia el servidor y del servidor hacia los diferentes clientes. Las principales ventajas de este modelo son:

1. **Buena escalabilidad**: El número de jugadores está limitado sólo por el ancho de banda del servidor y su capacidad de procesamiento.
2. **Buena consistencia**: Al solo haber una copia del estado del juego, es más sencillo mantener la consistencia entre los jugadores.
3. **Sincronización y administración simple**: Todo se encuentra en el servidor, por lo que sólo hay que preocuparse por mantener correctamente al mismo.
4. **Mayor facilidad para evitar las trampas**: Ningún jugador puede, localmente, modificar el mundo en su beneficio, pues todas las acciones de los jugadores son ejecutadas por el servidor.
5. **Facilidad de implementación**: Como analizaremos más adelante, implementar este modelo, frente a P2P, es muy sencillo.

Por otra parte, sus desventajas principales son:

1. **Punto central de fallo**: Si el servidor falla, el juego no puede continuar.
2. **Cuello de botella en el servidor**: Si el ancho de banda es limitado o la capacidad de procesamiento del servidor es baja, la experiencia de juego de los jugadores podría verse afectada.
3. **Es caro:** Aunque parezca obvio, los servidores hay que pagarlos y mantenerlos, convirtiéndose en un aspecto a tener muy en cuenta a la hora de definir los presupuestos.

4.3.2.2 Modelo Peer To Peer (P2P)

En este modelo no se diferencian actores o roles entre los clientes. Todos ocupan la misma posición jerárquica dentro del juego. El estado del juego es local a cada jugador, y se actualiza a partir de la información que se recibe desde cada uno de los otros jugadores. Por tanto, cada jugador, cada vez que realiza una acción, informa al resto de los jugadores, que deberán calcular localmente el nuevo estado de juego a partir esta información.

Modelo Peer To Peer

Durante una partida P2P, cada jugador se conecta con todos los demás jugadores, con los que intercambia constantemente información sobre sus acciones. Es gracias a esta información que puede calcular el nuevo estado del mundo y mantener la sincronización entre todos los jugadores. Si todos ejecutan las normas del juego de la misma manera, no deberían aparecer problemas de consistencia entre los jugadores, aunque tener la capacidad de ejecutar las normas de juego localmente puede permitir a los jugadores hacer trampas. Las principales ventajas de este modelo son:

1. **Múltiples copias del estado del juego:** En este caso se evita el punto central de fallo. Si un nodo de la red falla, el resto podrían seguir jugando.
2. **Menor coste de infraestructura:** En este caso, el estudio desarrollador no necesita correr con los gastos que conlleva un servidor, si no que los propios jugadores se comunican entre sí.

3. **Muy interesante para la distribución de contenido:** En muchos juegos, las comunidades de fans crean nuevo contenido para el juego en forma de mods, nuevos skins o mapas. Con este sistema, los jugadores podrían compartir este nuevo contenido sin ningún coste adicional.

Por otro lado, los contras de este modelo son:

1. **Baja escalabilidad:** El consumo de ancho de banda crece exponencialmente, pues cada nuevo jugador multiplica la cantidad de paquetes que hay que enviar y recibir.
2. **Inconsistente:** Algunos jugadores podrían no recibir algunos mensajes de actualización, quedándose con copias inconsistentes del estado del juego.
3. **Trampas:** Cada jugador es libre de realizar cálculos inconsistentes en su beneficio y transmitirlos al resto de jugadores, adulterando la partida.
4. **Dificultad de implementación:** Comparado con el modelo cliente-servidor, su implementación es mucho más complicada.
5. **Necesidad de un punto de encuentro:** Para que los jugadores puedan jugar entre sí, necesitan un lugar en el que poder encontrarse y lanzar las partidas.

4.3.3. Capa de Lobby

Las capacidades de red de un juego no deberían limitarse únicamente a aportar una plataforma en la que los jugadores puedan jugar juntos al mismo juego. Podrían utilizarse para mejorar la experiencia previa y alrededor del juego. Hay que tener en cuenta las comunidades que se generan alrededor de un juego muchas veces aportan tanto o más contenido que el creado inicialmente por el equipo de desarrollo. Hay también que darse cuenta que los jugadores necesitan un punto común en el que poder ponerse en contacto, no sólo para jugar, sino también para charlar y prepararse. No se puede olvidar que la red aporta unas posibilidades de socialización que obviamente no se pueden encontrar en la experiencia en solitario.

Entre los servicios más comunes que se suelen encontrar en la capa de lobby se encuentra el servicio de **autenticación**. Cada jugador necesita un nombre con el que poder ser reconocido por el resto de la comunidad. Este usuario puede ser independiente para cada juego, o puede formar parte de una plataforma mayor (como por ejemplo, Steam). El acceso mediante contraseña se vuelve especialmente importante si el juego contiene mecánicas relacionadas con la economía, como puede ser la compra de objetos en el juego.

Otro servicio de vital importancia para juegos orientados sobre todo a la competitividad entre los jugadores es el servicio de **MatchMaking.** Este servicio debe encargarse de poner en la misma partida a jugadores de un nivel de habilidad similar. Si las capacidades de cada jugador son muy diferentes la experiencia de los mismos se verá deteriorada: el novato perderá rápidamente y se sentirá frustrado, mientras que el experto sentirá que no hay un reto a su altura y se aburrirá.

Una solución común es utilizar algún sistema de rankings que se actualiza con cada partida. Por ejemplo, en *"FIFA World"*, se utiliza un sistema de liga dividido en 10 divisiones. El sistema de MatchMaking intentará emparejarnos con jugadores de nuestra liga o cercanas, pues deberían tener un nivel de habilidad similar al nuestro. Si se consigue una serie de

victorias en una división, se ascendería a la siguiente división, y si, por el contrario, una división es demasiado exigente para el jugador, este descendería a la división anterior. De esta forma se consigue no solamente que los partidos sean equilibrados, si no también generar en el jugador el deseo de mejorar y alcanzar la primera división.

Como ya se ha comentado, alrededor de los juegos más populares aparecen grandes **comunidades**. Estas comunidades no solo desean hablar del juego. Crean contenido, hacen amigos, comparten experiencias, negocian partidas… También buscan algún tipo de reconocimiento que haga saber a los demás jugadores que tan buenos son. Prestar atención a esta faceta del juego tendrá siempre un efecto positivo, pues son las comunidades de jugadores las que son capaces de alargar la vida de un videojuego hasta límites insospechados. Ejemplo de ello es *"StarCraft"* de Blizzard, que, lanzado en 1998, y con su secuela ya en el mercado, se mantiene en el top de videojuegos jugados en Corea del Sur, siendo juego pionero de lo que actualmente se conoce como e-Sports.

4.3.4. Problemas

El desarrollo de videojuegos en red, por si fuera poco, además sufre de una serie de males endémicos que deben ser analizados en fase de diseño, y tratados con el mayor de los cuidados en fase de implementación. Algunos ejemplos son:

Consumo del ancho de banda: Se produce cuando la capacidad de la infraestructura no es suficiente para soportar la cantidad de datos que se envían, o cuando las entidades no han contratado con su respectivo ISP suficiente velocidad o potencia de red.

Para paliar este problema, se puede optar por distintas soluciones. El uso de una **arquitectura cliente-servidor** es bastante eficaz, pues reduce la cantidad de paquetes a manejar por las distintas entidades de la red. La **reducción del tamaño** de los paquetes junto con un **envío selectivo** de los mismos (enviar solo datos que hayan cambiado) se presentan como técnicas bastante interesantes para reducir el consumo del ancho de banda. Otra solución es reducir intencionadamente la frecuencia de envío e implementar técnicas de predicción de movimiento en los clientes. Entre las técnicas de predicción destaca **Dead Reckoning**, que analizaremos más adelante en este mismo documento.

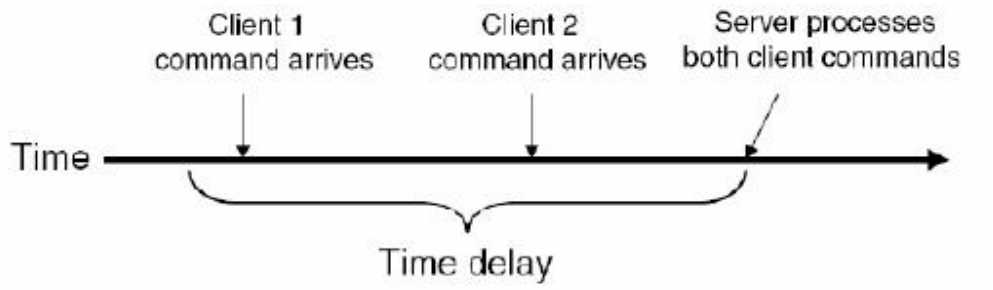
Latencia: La latencia es el tiempo que tarda en llegar la información a los jugadores. Cuando esta latencia es demasiado alta es conocida popularmente como LAG y es la principal razón para el enfado y hastío de los jugadores. La latencia depende de varios factores, entre los que se encuentran el tiempo que se tarda en **calcular el nuevo estado del juego**, el tiempo que tarda el protocolo de red en **poner el mensaje en la red,** las técnicas de **compresión, descompresión, encriptación y desencriptación** de los paquetes, retardo de la **transmisión por la red…**

De verdad, muy violento.

Una latencia demasiado alta puede tener efectos desastrosos en la experiencia de juego, pudiendo generar inconsistencias en el estado del juego, y dar ventaja a los jugadores con un menor LAG, ya que, al recibir y contestar a los mensajes antes, pueden tomar decisiones con antelación al resto de jugadores. Las latencias superiores a los 100 ms no son toleradas por casi ningún juego.

Uno de los mayores expertos en estas lides es Yahn W. Bernier, trabajador de Valve y autor de *Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization* [9]. Una posible solución que propone Bernier al problema de la latencia es el uso de una filosofía **Peer 2 Peer**, ya que cada jugador recibe actualizaciones con mayor asiduidad, ya que es informado por todos los jugadores de la red. El uso del **protocolo UDP** es preferible al uso de TCP, pues reduce en gran parte los tiempos de compresión, descompresión, transmisión… Los problemas de consistencia podrían ser solucionados mediante las técnicas de **Time Delay** y **Time Wrap.**

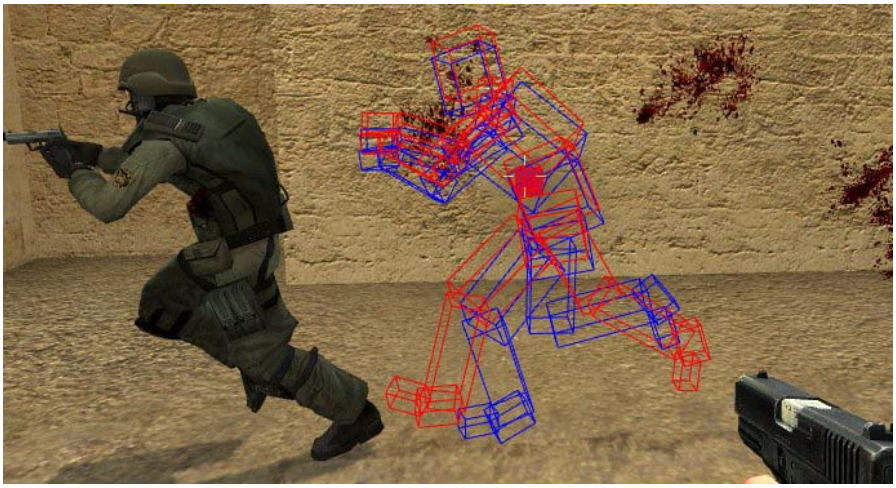
Usando **Time Delay**, el servidor esperaría a recibir los comandos de todos los jugadores antes de calcular el siguiente estado del juego, evitando así que un jugador tenga ventaja sobre el resto, pero retrasando el cálculo del estado de juego, castigando a los jugadores con mejores conexiones.



Funcionamiento del Time Delay

La técnica de **Time Wrap** consiste, por otra parte, en realizar los cálculos del estado de juego para el estado del mundo correspondiente al momento en el tiempo en que el comando del jugador fue enviado. Por ejemplo, en un juego de disparos, si, a razón del LAG, un disparo

que en el instante de tiempo en que fue realizado iba a ser certero es calculado como errado, la técnica de Time Wrap corregiría este error y realizaría la acción necesaria por el impacto correcto.



Time Wrap en Counter Strike

Finalmente, el gran problema al que puede enfrentarse un desarrollador de videojuegos online, así como sus jugadores más honrados, son las trampas. Así como durante la experiencia en solitario las trampas no afectan a la experiencia del jugador, pues este es consciente de que las está realizando, durante una partida multijugador hay que tener en cuenta que las decisiones de un jugador afectan a los demás, y si decide tomar ventaja por medios poco lícitos enturbiará la experiencia de juego de los demás jugadores. Algunos ejemplos de trampas que los jugadores pueden utilizar son:

LAG Artificial: El jugador tramposo insufla LAG de manera artificial en el juego, reduciendo en gran cantidad el número de paquetes que envía al resto de jugadores. Se utiliza sobretodo en juegos P2P. El efecto que produce es que los demás jugadores reciben pocas actualizaciones del jugador, viendo como parece teletransportarse por el escenario, mientras que el tramposo puede ver a los demás jugadores con normalidad.

Auto apuntado: Mediante el uso de software externo, el jugador tramposo aprovecha que recibe información sobre la posición de los demás jugadores aunque no los esté viendo. Conociendo esta información, apuntar se convierte en una trivial que solo necesita calcular la posición de un enemigo con respecto a la posición del tramposo en cuestión.

Wallhacking: Consiste en manipular las texturas de los objetos haciéndolas transparentes o semitransparentes, o incluso modificar el mapa creando agujeros en los modelos, de forma que el jugador pueda ver más allá de las paredes, pudiendo ver a los jugadores rivales de manera anticipada.

Existen más tipos de trampas, muchas de ellas no relacionadas con la tecnología, si no con la falta de deportividad de los jugadores, desconexiones intencionadas para evitar una derrota, creación de varias cuentas en un mismo juego para enfrentarse a uno mismo y escalar puestos en un sistema de ranking… Cada tipo de trampas debe ser analizada y tratada por

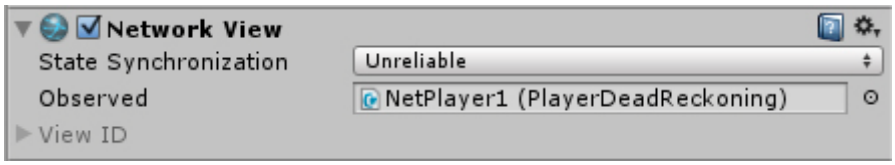
separado, aunque algunas compañías han desarrollado software de prevención de trucos como GameGuard o VAC.

4.4. Unity3D para juegos en Red

El motor Unity3D es, como ya hemos analizado, un interesante, potente y económico motor para el desarrollo de videojuegos. Además, provee servicios especializados para el desarrollo de videojuegos en red que afectan de manera sensible a los mismos. La documentación de Unity ya mencionada en este documento, así como el libro *Unity Multiplayer Games*[7] ofrecen una base interesante a la hora de afrontar el reto de la implementación de juegos en red en Unity.

Por una parte, Unity3D no soporta el concepto de arquitectura P2P de manera nativa. Exige la existencia del rol de servidor. Sin embargo, es posible desarrollar juegos con el modelo P2P simplemente tratando al servidor como un jugador más, y utilizando de forma inteligente el mecanismo de Remote Procedure Calls o llamada a procedimientos remotos (RPC). Este mecanismo será analizado con mayor detalle en la sección correspondiente a la implementación del modelo P2P en Beat Fighters.

Para la comunicación y sincronización de los distintos gameObjects a través de la red, Unity3D proporciona un componente llamado **NetworkView**. Todo gameObject que tenga este componente, automáticamente obtiene la capacidad de realizar RPC's, así como sincronización de estado. Incluso es capaz de sincronizar automáticamente otro componente a través de la red.



Componente NetworkView

Cada NetworkView puede configurarse de manera independiente para que utilice el protocolo UDP o el RUDP, según la necesidad, o incluso puede desconectarse su capacidad de sincronización y ser usado únicamente para realizar las llamadas a RPC's. También se puede enviar paquetes personalizados mediante script. Esta capacidad la analizaremos en detalle en el desarrollo del modelo cliente-servidor de Beat Fighters.

El equipo de Unity3D ofrece a los desarrolladores una plataforma en la que poder registrar las partidas de un juego para hacerlas públicas, sirviendo de punto de encuentro para los jugadores. Esta plataforma se llama **MasterServer**. Aunque su nombre puede llevar a engaño, el MasterServer no es un servidor de juego. Únicamente actúa como tablón de anuncios para las distintas partidas que estén esperando jugadores. Pone en contacto a los distintos equipos e incluso es capaz de resolver los problemas de DHCP que puedan surgir.

En el desarrollo de Beat Fighters nos hemos apoyado del MasterServer que proporciona Unity3D, aunque no hemos podido montar nuestro propio MasterServer, pues requiere tener una IP pública, de la cual no se disponía en el momento de realizar este proyecto.

Finalmente, Unity3D nos permite configurar la cantidad de paquetes que serán enviados por segundo. Por defecto, está configurado para enviar 15 paquetes por segundo, pero para el desarrollo de Beat Fighters se ha reducido este número a 7, para acentuar el efecto de la técnica de Dead Reckoning.

4.5. Redes en Beat Fighters

4.5.1. Capa de Lobby

A la hora de desarrollar la capa de red de Beat Fighters, y antes de adoptar cualquier modelo, el primer paso fue conseguir que los jugadores pudieran conectarse entre sí. Para aprovechar la existencia del MasterServer de Unity3D antes mencionado, se optó por que una de las entidades de la red, bien el servidor en el modelo cliente-servidor o uno de los clientes del modelo P2P creara un servidor y registrara la partida en el MasterServer. Recordemos que Unity3D no permite P2P de forma nativa, por lo que uno de los jugadores debe tener el rol de servidor, aunque no realice ninguna tarea de servilismo real.

```
public void StartServer()
{
    Network.InitializeServer(2, 25001, !Network.HavePublicAddress());
    MasterServer.RegisterHost(gameUniqueName, "LobbyName",
                              "Beat FightersGameRoom");
}
```

Para encontrar una partida, el cliente solo necesita hacer una solicitud al MasterServer pasándole el identificador del juego. De forma asíncrona, el servidor contestará con la lista de partidas disponibles. Una vez recibida, el jugador puede conectarse a cualquier partida que tenga aún espacio para otro jugador. En el fragmento de código superior, en el método InitializeServer se especifica, como primer parámetro, el número de jugadores que admite la partida. Es importante tener en cuenta que el servidor no cuenta como jugador, por lo que, en este caso, un 2, representa que puede haber 2 jugadores además del ente servidor. Por tanto, en la versión P2P de Beat Fighters sólo admitiremos un jugador extra, pues para nosotros, el servidor es un jugador más. El método OnMasterServerEvent es llamado cada vez que el MasterServer quiere comunicarse con la aplicación.

```
private void OnMasterServerEvent(MasterServerEvent msEvent)
{
    switch (msEvent)
    {
        case MasterServerEvent.RegistrationSucceeded:
        ...
        case MasterServerEvent.HostListReceived:
        ...
        hostData = MasterServer.PollHostList();
        ...
    }
}
```

En las clases AuthoritativeServerManager y P2P_ServerManager se encuentra tanto este tratamiento como el realizado cada vez que un jugador se conecta a la partida. Este último comportamiento será analizado en las secciones correspondientes.

4.5.2. Modelo Cliente – Servidor

4.5.2.1 Aspectos Generales

La implementación del modelo cliente – servidor se convierte, si ya se ha desarrollado la mayor parte del juego sin pensar en red, en una cuestión casi trivial. Solo hay que decidir dónde se ejecutará cada parte del código. Las normas del juego, como el movimiento, el control del flujo del juego, la instanciación de objetos, la comunicación con los clientes y la sincronización deben ser ejecutados por el servidor. El cliente, por su parte, debe ser capaz de enviar al servidor las acciones que debe realizar, analizar las respuestas del servidor y compensar la latencia para una experiencia de juego fluida.

Para distinguir de forma unívoca dónde debe ejecutarse cada clase sin la necesidad de crear dos proyectos distintos para el cliente y el servidor, se han creado dos clases: **ServerScript** y **ClientScript.** Estas clases harán de padres para el resto de clases. Su única función será la de identificar si se encuentran en el lado correcto de la red y, en caso de estarlo, mantener el componente activo o, en caso contrario, deshabilitarlo.

```
public class ClientScript : MonoBehaviour {
    . . .
    protected void OnEnable()
    {
        if (Network.isServer)
            this.enabled = false;
    }
    . . .
}
```

De esta manera, nos encontramos con que el único script que se ejecutará únicamente en cliente es la clase **C_KeyListener** encargada de hacer la escucha desde teclado y enviar los comandos al servidor. Nótese que esta clase está emparejada con la clase **S_KeyListener**, que es la encargada de reflejar los comandos enviados por el cliente en acciones en el servidor.

Ahora bien, este componente se encuentra en ambos personajes, y necesitamos distinguir a cual personaje debe mover cada cliente. Unity3D nos ofrece el atributo **NetworkView.IsMine.** El problema es que considera dueño de un objeto a quien lo haya instanciado. En nuestro caso, todas las instancias se realizan en el servidor, por tanto, para Unity3D, el dueño de todos los gameObjects es el servidor.

Este problema es solventado en el momento en que un jugador se une a la partida. En ese momento, se instancia un gameObject jugador para este nuevo cliente, y se hace una llamada RPC al método **SetOwner** en la clase **C_KeyListener** para asignar el gameObject en cuestión al nuevo jugador.

```
public class AuthoritativeServerManager : MonoBehaviour {

        private void SpawnPlayer(NetworkPlayer netPlayer)
        {
            GameObject player = Warehouse.Instance.InstantiateObject(playerGOs[players],
                                        spawnPositions[players].position);
            player.GetComponent<NetworkView>().RPC("SetOwner", RPCMode.AllBuffered,
                                                   netPlayer);
            players++;
        }
}
```

En el fragmento de código anterior puede verse tanto el comportamiento previamente descrito, como la existencia de una clase llamada **Warehouse** que se encarga de la instanciación. Esta clase, que sigue el **patrón Singleton** se ejecuta únicamente en servidor y es la encarga de instanciar y destruir todos los gameObjects del juego, para un mejor control de los mismos dentro de la red.

Como detalle final, comentar que, la clase C_KeyListener sólo envía información si esta ha cambiado. Es decir, si se ha producido una pulsación nueva. De esta manera, evitamos congestionar la red con información redundante.

```
void Update()
{
    . . .
    if (IsMine())
    {
        float horizontal = Input.GetAxis("Horizontal_P1");
        float vertical = Input.GetAxis("Vertical_P1");

        if (inputChanged(horizontal, vertical))
        {
            networkView.RPC("UpdateMotion", RPCMode.Server,
                            horizontal, vertical);
            lastHorizontal = horizontal;
            lastVertical = horizontal;
        }
    }
}
```

4.5.2.2 Sincronización del Estado de Juego.

En el modelo cliente-servidor, toda la lógica de juego se ejecuta en el servidor. Por tanto, es este el que calcula las nuevas posiciones de los jugadores. Estas posiciones son transmitidas a los jugadores para que el estado del juego se mantenga consistente.

En una primera aproximación, nos limitamos a asignar el componente Transform de cada gameObject al atributo observado del componente NetworkView. De esta manera conseguimos sincronizar tanto la posición como la escala de las distintas entidades de juego. El problema resultó ser que, como habíamos configurado, sólo se recibían 7 paquetes por segundo. Esto significa que actualizamos la posición 7 veces por segundo, un ratio muy inferior a los al menos 30 frames por segundo que se necesita para obtener un movimiento fluido. Por ello, se optó por una segunda solución: definir nuestros propios paquetes de red. De esta manera podremos manejar la información que nos llega y realizar procedimientos de interpolación y Dead Reckoning.

Nuestros paquetes contendrán la información necesaria para poder mantener un estado de juego consistente, pero a la vez fluido para el jugador. Utilizaremos como ejemplo la sincronización del personaje jugador.

Para la definición de los paquetes propios en Unity3D hay que sobrescribir el método OnSerializeNetworkView. En este método podemos captura el bitstream que va a ser enviado o recibido y modificar la información a nuestro gusto. En el caso del jugador, se decidió enviar la información sobre la posición del jugador y su orientación para una correcta colocación del

personaje y su velocidad para poder realizar la predicción del movimiento, utilizando la técnica de Dead Reckoning.

```
public override void OnSerializeNetworkView(BitStream stream,
                                             NetworkMessageInfo info)
{
    Vector3 syncPosition = Vector3.zero;
    Vector3 syncVelocity = Vector3.zero;

    if (stream.isWriting)
    {
        syncPosition = this.transform.position;
        stream.Serialize(ref syncPosition);

        syncRotation = this.transform.rotation;
        stream.Serialize(ref syncRotation);

        syncVelocity = new Vector3(this.rigidbody2D.velocity.x,
                                   this.rigidbody2D.velocity.y, 0.0f);
        stream.Serialize(ref syncVelocity);
    }
    else
    {
        stream.Serialize(ref syncPosition);
        stream.Serialize(ref syncRotation);
        stream.Serialize(ref syncVelocity);

        CreateClone(syncPosition);
        RestartSyncTime();
        CalculatePositions(syncPosition, syncVelocity);
    }
}
```

Una vez recibida la información en el cliente, se nos abren tres frentes. Por una parte, podemos asignar la información directamente en el gameObject. De esta manera, conseguimos el mismo efecto que teníamos al asignar el componente Transform en el NetworkView.

La segunda de las opciones disponibles es la de **interpolar** las posiciones. Al recibir la nueva posición en la red, se simula el movimiento hasta que alcanzar esta nueva posición. De esta manera eliminamos los saltos por la falta de paquetes, consiguiendo un movimiento fluido. El inconveniente es que lo que el jugador ve siempre está "en el pasado" es decir, siempre juega con un cierto retraso con respecto al servidor.

La tercera opción consiste en **predecir** el movimiento. Los cambios en la dirección del movimiento normalmente no son bruscos, de un frame a otro, si no que el movimiento suele ser continuado durante algunos segundos. Por ello, conociendo la posición actual del personaje, y su velocidad (si es constante) o su aceleración (si la velocidad no lo es), podríamos calcular, de ante mano, hacia dónde se dirige. Para ello, se utilizarían las fórmulas de física clásica para el movimiento rectilíneo uniforme:

$$MRU: x = x_0 + vt$$

$$MRUA: x = x_0 + vt + \frac{1}{2}\,at^2$$

En el código, queda de la siguiente manera, siendo syncDelay el tiempo transcurrido entre los dos últimos paquetes recibidos:

```
private Vector3 PredictNextPosition(Vector3 syncPosition, Vector3 syncVelocity)
```

```
{
    return syncPosition + syncVelocity * syncDelay;
}
```

Esta técnica es conocida como **Dead Reckoning**. Añadir que en el momento de elegir entre UDP y RUDP, se optó por UDP, pues necesitamos ante todo velocidad. La posición y velocidad del gameObject van a cambiar a la suficiente velocidad para que la pérdida de un paquete o un paquete corrupto afecten a la jugabilidad.

Dentro de la ejecución de Beat Fighters, se puede intercambiar entre los tres modos de análisis del paquete recibido. Para la versión interpolada y para la predictiva, además, aparecerá un sprite semitransparente del personaje, que se colocará en la posición recibida en el paquete. De esta manera, podremos ver como en la versión interpolada, el "fantasma" se coloca siempre delante del jugador, en la posición hacía la que se dirige. Sin embargo, en la versión predictiva, el "fantasma" queda siempre por detrás del jugador, pues es precisamente el adelantarnos a lo que está por venir el efecto que buscamos. Se aprecia, también, como cuando la predicción es errónea (el movimiento ha cambiado entre los dos últimos paquetes recibidos) el personaje corrige su posición con un frame de retraso.

4.5.3. Modelo Peer 2 Peer

4.5.3.1. Aspectos Generales

A la hora de implementar la versión Peer 2 Peer de Beat Fighters, hubo que realizar sensibles cambios en la implementación del juego. Esto es debido a que hay código que se ejecutará en todos los clientes, mientras que otra parte depende únicamente del jugador dueño de un cierto gameObject.

Ante la tesitura de decidir qué jugador es dueño de que objetos, esta vez sí podemos utilizar la potencialidad de Unity3D y aprovechar el atributo *NetworkView.IsMine*. Esto se consigue simplemente indicando a cada jugador que es responsable de instanciar a su personaje. Para ello, utilizamos el método *OnConnectedToServer()*, que es llamado en el cliente en el momento en que se conecta al servidor de juego, para realizar la instanciación.

El problema surge cuando debemos decidir a quién pertenecen los objetos que vienen a formar la lógica de juego, como el profesor o los instrumentos. En nuestra implementación, hemos optado por asignarle estos objetos al primer jugador que se une a la partida. Esto es por el simple hecho de elegir a uno arbitrariamente, pues bien podríamos haber elegido a cualquier otro jugador, o incluso dividir los objetos, de manera que uno de los jugadores controlara los marcadores, mientras otro controla los instrumentos, por ejemplo. Cada vez que un jugador necesite que alguna acción sea realizada por otro objeto, o incluso los objetos entre sí, harán llamadas RPC en busca del cliente que es dueño del objeto, que realizará la acción.

Por ejemplo, cada vez que un jugador coloca un instrumento en el pentagrama, se ha de comprobar si con este instrumento acaba la partida. Para ello, el propio pentagrama realizará una llamada RPC en busca del cliente dueño del controlador del nivel, que evaluará si la partida debe darse por finalizada y actuará en consecuencia.

```
void OnTriggerEnter2D(Collider2D other)
{
    if (CanBeCatched(other))
    {
        networkView.RPC("DiscardActualInstrument", RPCMode.All);
        networkView.RPC("AttachInstrument", RPCMode.All,
                        other.gameObject.networkView.viewID);
        levelController.networkView.RPC("CheckVictory", RPCMode.All);
    }
}
```

En esta implementación, debemos distinguir entre que va a ser ejecutado únicamente por el dueño de un objeto, y que va a ser ejecutado por todos los jugadores. Para ello, y siguiendo la filosofía que utilizamos en la arquitectura cliente-servidor, se ha implementado una clase, llamada **OwnerScript**, de la que heredarán todas las clases que solo deben ser ejecutadas por el dueño de un objeto concreto.

```
public class OwnerScript : MonoBehaviour {

    protected void OnEnable()
    {
        if (!networkView.isMine)
        {
            this.enabled = false;
            DisableCollider();
        }
    }
}
```

Añadir que, en este caso, si un objeto no nos pertenece, se deshabilita también cualquier componente físico que pudiera tener, para evitar así que se cada jugador realice sus propios cálculos de colisión para objetos que no le pertenecen, transmitiéndolos a través de la red y generando inconsistencias.

Para distinguir que debe ejecutarse en cada cliente, hemos decidido distinguir entre **decisiones** y **acciones.** Una acción significaría realizar algún tipo de movimiento que cambia el estado del mundo. Una decisión, es el razonamiento que lleva a la acción. Por tanto, es obvio que las decisiones solo pueden ser tomadas por el dueño de un gameObject. Este, decidirá qué acción realizará a continuación, realizándola e informando de ello al resto de jugadores, que también la ejecutarán en sus representaciones del jugador. De esta manera, podríamos tomar como ejemplo, nuevamente, el control del objeto.

Un jugador tomaría sus decisiones en función de lo que sucede en la pantalla. A continuación, pulsaría o retiraría una o varias teclas. Este procedimiento debe ser capturado únicamente por el personaje que representa al jugador en el juego.

```
public class P2P_PlayerDecisions : OwnerScript {

    public int nPlayer = 1;

    void Update()
    {
        if (IsThrowPressed())
            networkView.RPC("ThrowInstrument" , RPCMode.All);

        if (IsDiscardPressed())
            networkView.RPC("DiscardInstrument", RPCMode.All);

        DoMovementPressed();
    }
}
```

Una vez tomada la decisión, se realiza la llamada RPC a la acción correspondiente. Las acciones son ejecutadas por todos los jugadores de la red cuando reciben el mensaje, manteniéndose de esta manera la sincronización entre ellos.

```
public class P2P_PlayerActions : MonoBehaviour {
    ...
    [RPC]
    public void ThrowInstrument()
    {
        if (HasInstrument())
        {
            attachedInstrument.GetComponent<P2P_InstrumentController>()
                                        .ThrowInstrument();
            attachedInstrument = null;
        }
    }
    ...
}
```

4.6. Conclusiones

Si el desarrollo de videojuegos es, de por sí, un proyecto complejo, la inclusión de mecánicas de red lo convierte en algo aún más complicado. La decisión de desarrollar un juego online no debe ser tomada a la ligera. Como hemos podido aprender durante el desarrollo de Beat Fighters, no se trata de una tarea puramente de programación, si no que compete a todas las áreas del equipo de desarrollo, con especial peso en la etapa de diseño.

Las alternativas de implementación son variadas. Existe una gran cantidad de modelos y tecnologías que pueden ser usadas, amén del posible desarrollo de soluciones híbridas según la situación del juego. Un ejemplo a voz de pronto: la capa de lobby podría usar TCP mientras que el juego en si sería UDP. Es importante analizar previamente que tipo de solución es la que mejor se ajusta al diseño del juego, sobretodo porque debemos tener en cuenta que el desarrollo en red conlleva una serie de gastos adicionales, como podría ser la contratación de un servidor, que no todos los equipos de desarrollo pueden permitirse.

En el caso de Beat Fighters, a priori la mejor opción hubiese sido un desarrollo orientado al modelo P2P. Se trata de un juego de pequeñas dimensiones, en el que la carga de paquetes sobre la red no va a ser excesiva y el número de jugadores es bajo, por lo que no debería producirse un consumo excesivo del ancho de banda. Además, al ser un juego orientado a la competición entre dos jugadores, el juego podría alargar su vida útil muchos más años, pues los propios jugadores podrían jugar en cualquier momento sin necesidad de mantener una infraestructura por parte nuestra.

Sin embargo, el desarrollo P2P supuso cambios bastante profundos en el desarrollo que ya se había realizado previamente. Al haberse desarrollado la mayor parte del juego en una etapa temprana del proyecto, y no haberse planteado, en ese momento, la orientación a P2P, tuvimos que corregir muchos aspectos del desarrollo para adaptarlo a la nueva arquitectura. Esto en un desarrollo real podría ser un gran contratiempo, pues significaría invertir tiempo de desarrollo y dinero en rehacer gran parte del juego. Por tanto, el modelo cliente-servidor podría convertirse en una solución interesante para dicha situación, pues los

cambios en la implementación no serían tan graves. Pero tendría el contra de tener que contratar servidores.

Algunas mejoras que se podrían incluir para expandir el juego podría ser el añadido de un chat, tanto escrito como de voz para mejorar la comunicación entre los jugadores. Esta mejora sería interesante para ambas arquitecturas. Por otra parte, plantear un sistema de pausa, sobre todo a nivel de diseño podría ser muy interesante a la hora de incluirlo en la plataforma online.

5. Conclusiones y Trabajo Futuro

"El primer 90% del código corresponde al primer 90% del tiempo de desarrollo.
El 10% restante corresponde al otro 90% del desarrollo"
Regla del 90-90 de Tom Cargill, Bell Laboratories, 1985

5.1. Conclusiones

Finalmente, y después de meses de duro esfuerzo, llega la hora de analizar el trabajo realizado. Por una parte, la realización de este proyecto me ha permitido trabajar de una manera de la que nunca antes había trabajado. El proyecto se puede resumir como una demo técnica, dónde lo más importante ha sido la obtención de experiencia y conocimiento sobre las distintas técnicas. El objetivo no era implementar la opción más óptima para cada situación desde el principio, si no probar las distintas posibilidades y descubrir mediante la experiencia las ventajas y desventajas de cada técnica, así como su impacto en el resultado del juego.

Por otra parte, considero que el haber realizado este proyecto utilizando el motor de videojuegos Unity3D ha sido un gran beneficio para mí como profesional. Muchas empresas del sector reclaman profesionales con conocimientos de dicho motor. Además su relativamente bajo precio, y la posibilidad de publicar de forma gratuita para la mayoría de dispositivos me permiten incluso pensar en la posibilidad de montar mi propio estudio, pues la inversión inicial no sería tan agresiva como aparentemente podría ser.

Finalmente, el desarrollo de este documento también ha sido aportado un importante crecimiento a mi persona, pues me ha obligado a tratar de ser lo más claro y conciso posible, buscando siempre el llegar a la mayor cantidad de público posible, sin descuidar los aspectos técnicos de los que se hablaba en cada momento.

5.2. Trabajo Futuro

Como siempre que se realiza un trabajo de estas características, queda siempre mucho que mejorar. En este caso concreto, existen dos líneas de trabajo principales perfectamente diferenciadas:

Crecimiento de la demo

En el proyecto se han desarrollado técnicas de juego en red y de inteligencia artificial. Sin embargo, en el desarrollo de videojuegos existen muchas más facetas técnicas dignas de análisis. La programación de shaders y efectos de post-procesado se presenta como una alternativa interesante a analizar. Otra alternativa podría ser la implementación de Beat Fighters en 3D, y analizar las diferencias con el modelo 2D. Distintos modelos de físicas podrían significar también una diferencia notable en la jugabilidad.

En resumen, el desarrollo de videojuegos es un campo multidisciplinar, del cual en el proyecto se han visto reflejado dos de los principales campos, pero que podría extenderse hasta abarcarlos todos.

Desarrollo real de Beat Fighters

Beat Fighters es, ante todo, un juego. Durante el desarrollo del proyecto ha estado acotado a unas pocas mecánicas para mantener la complejidad del proyecto en límites aceptables. Ahora, con la experiencia escrita que supone este documento, podría plantearse

el afrontar un desarrollo real del juego con el objetivo de publicarlo y permitir que jugadores de todo el mundo disfruten la experiencia que supone Beat Fighters. Personalmente, haría un especial esfuerzo en el desarrollo del diseño, añadiendo mecánicas que hagan al juego más dinámico.

Anexos

“It’s Me! Mario!”
Super Mario.

ANEXO 1: Manual

Beat Fighters es al mismo tiempo, un juego multijugador y una demo técnica de distintas técnicas utilizadas en el desarrollo de videojuegos. Para inicial el juego, no es necesaria instalación alguna, solamente lanzar el ejecutable que se proporciona junto a este manual.

Beat Fighters como juego

Al empezar una partida en Beat Fighters, los jugadores escucharán una melodía compuesta por distintos instrumentos, en un orden específico. El objetivo de los jugadores será el de recomponer el pentagrama que han escuchado, utilizando para ellos una serie de instrumentos que caerán desde la parte superior del escenario. El primer jugador en completar su pentagrama será el ganador.

Los controles para el jugador 1 son los siguientes:

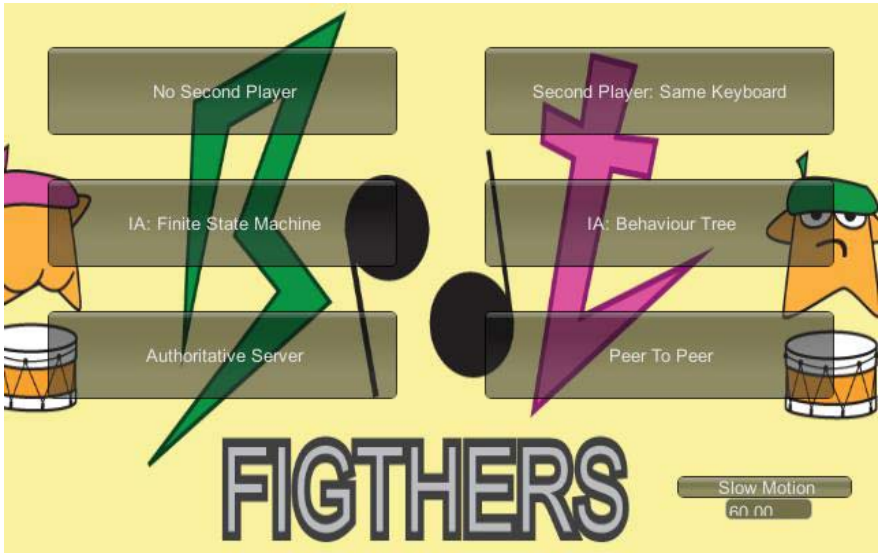
- **A - D:** Moverse a Izquierda / Derecha
- **W:** Lanzar un instrumento recogido hacia los pentagramas.
- **S:** Descartar un instrumento recogido.

Para el jugador 2, en el modo de juego en el mismo teclado, son los siguientes:

- **Flecha Izquierda – Flecha Derecha:** Moverse a Izquierda / Derecha
- **Flecha Arriba:** Lanzar un instrumento recogido hacia los pentagramas.
- **Flecha Abajo:** Descartar un instrumento recogido.

Beat Fighters como demo

Beat Fighters se compone de 6 implementaciones distintas del mismo juego. Para todas las implementaciones, existe un botón de ***Slow Motion*** que permite observar el funcionamiento del juego a cámara lenta, especialmente útil para las secciones de inteligencia artificial.



Menú Principal

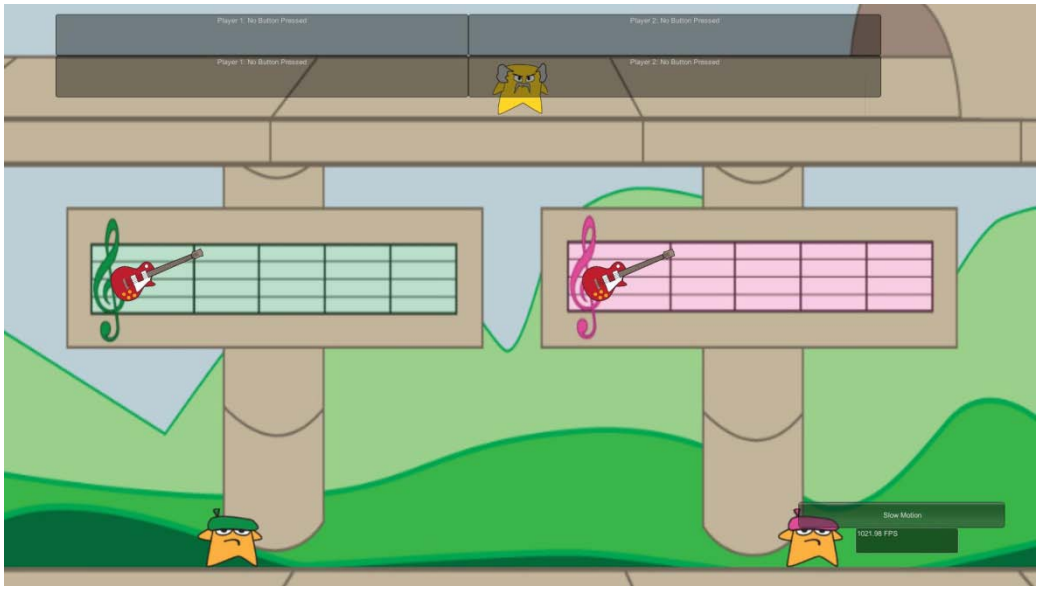
Beat Fighters

Esta implementación permite al usuario jugar una partida sin rival, observando el esqueleto básico del juego.

Mismo Teclado

Esta implementación, accesible a través del botón "***Second Player: Same Keyboard***" permite al usuario jugar una partida contra un rival humano, usando ambos el mismo teclado.

En la parte superior de la pantalla puede observarse en todo momento, que pulsaciones están siendo realizadas en cada momento por cada jugador.



1

Inteligencia Artificial: Máquinas de Estado

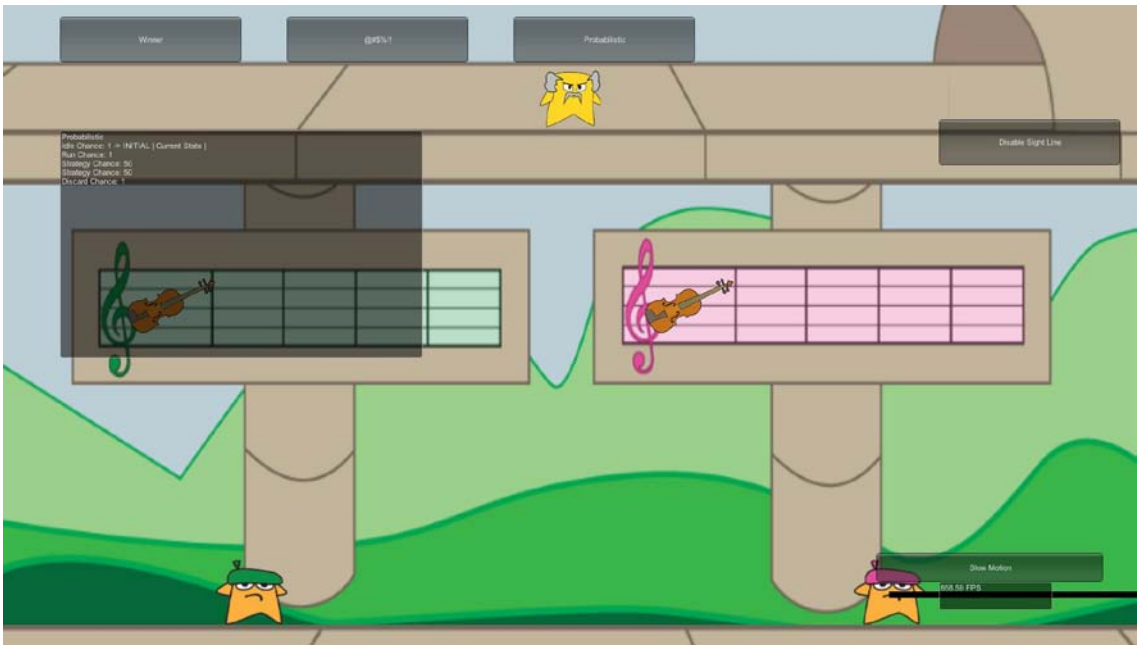
A esta implementación se accede mediante el botón "**IA: Finite State Machine"** del menú principal. Permite al usuario jugar una partida contra un rival controlado por la inteligencia artificial, basada en máquinas de estado.

Existen 3 comportamientos disponibles, intercambiables mediante los botones de la parte superior de la pantalla, siendo estos comportamientos:

- **Mentalidad Ganadora** *(Botón Winner):* La inteligencia artificial intentará ganar la partida.
- **Mentalidad No Perdedora** *(Botón @#$%!!):* La inteligencia artificial intentará evitar que el jugador humano gane la partida.
- **Mentalidad Mixta** *(Botón Probabilistic):* La inteligencia artificial decidirá en cada momento, de forma aleatoria, si jugará para ganar o para incordiar al rival.

En todo momento puede verse en pantalla un cuadro con los estados que componen la máquina activada, resaltado el estado activo en cada momento.

Además, y de manera común a las dos implementaciones basada en inteligencia artificial, se muestra una línea que representa la línea de visión del personaje no jugador. Esta línea puede deshabilitarse mediante el botón ***"Disable Sight Line"***.

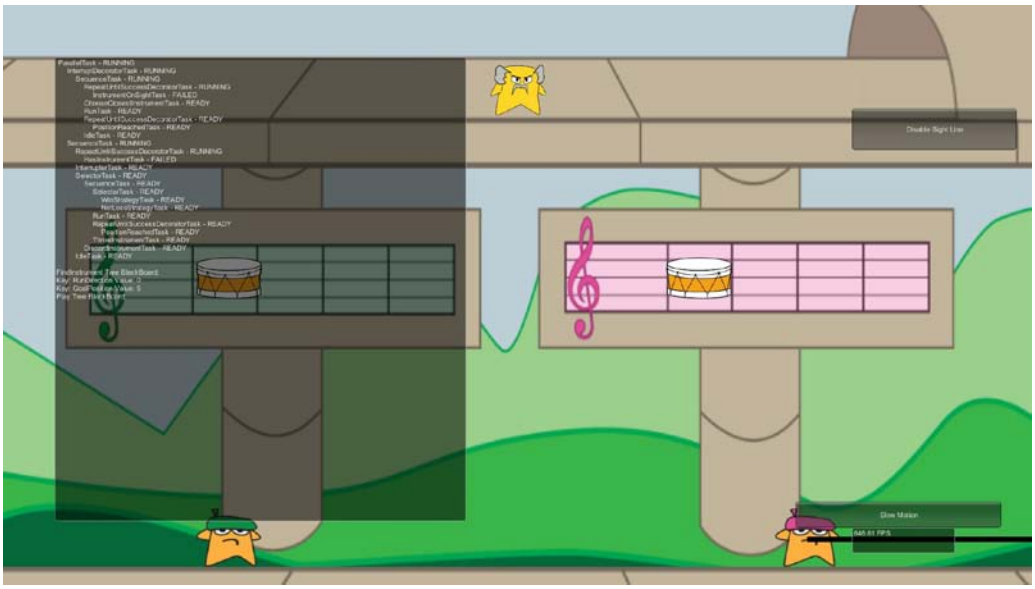


Máquinas de Estado

Inteligencia Artificial: Árboles de Comportamiento

A través del botón ***"IA: Behaviour Tree"*** se llega a esta implementación, que permite al usuario jugar una partida contra un rival controlado por una inteligencia artificial basada en árboles de comportamiento.

En esta ocasión, en la pantalla se podrá observar en todo momento una descripción del estado del árbol, con todas las tareas y su estado en cada momento. Además, también se muestran los datos escritos en las pizarras.



Árbol de Comportamiento

Network: Modelo Cliente-Servidor

Esta implementación, a la que se accede con el botón ***"Authoritative Server"*** del menú principal, permite a dos jugadores jugar una partida competitiva a través de la red.

Al entrar en esta implementación, podrán tanto crear un nuevo servidor ***(Botón: "Create Server")***, como unirse a una partida ya existente ***(Botón: "Find Servers")***. Para poder iniciar una partida, es **necesario** que exista un servidor en funcionamiento. Por tanto, **la partida requiere 3 instancias** de *Beat Fighters* en ejecución: dos clientes y un servidor. Estas instancias pueden estar en equipos diferentes.

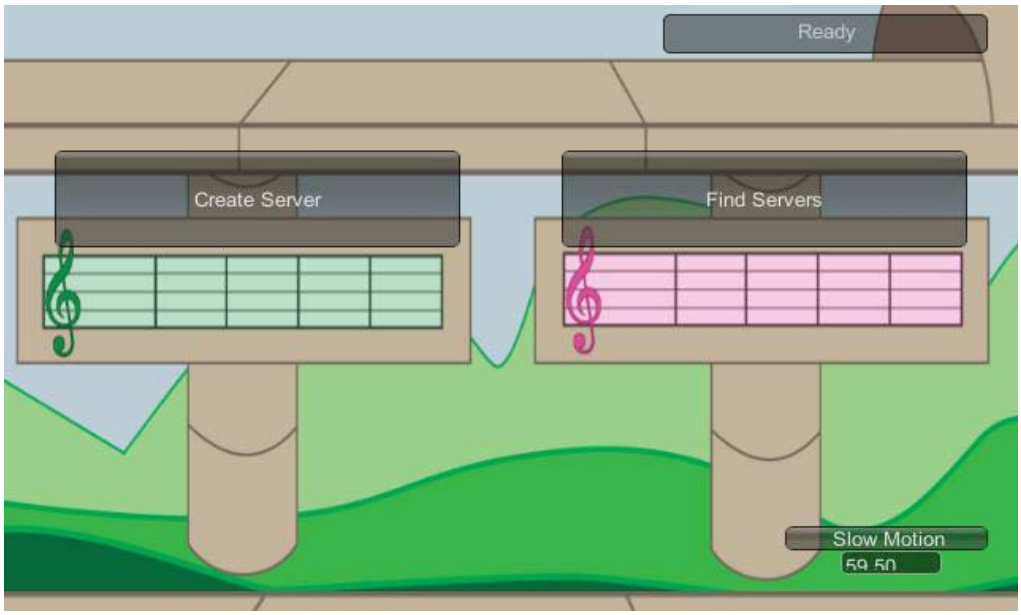
Una vez iniciado un servidor, los jugadores pueden conectarse a través de la misma pantalla. Al presionar el botón ***"Find Servers",*** aparecerá, de existir, una nueva lista de botones con los nombres de las partidas. Para conectarse a una, sólo hay que oprimir el botón correspondiente. Cuando haya dos jugadores en el mismo servidor, la partida se iniciará automáticamente. A partir de este momento, cada jugador puede decidir como visualizar la sincronización de su personaje. Para ello, puede elegir entre tres opciones:

1. **Básico** *(Botón Basic)*: Cada vez que se reciba un paquete de red, se actualiza la posición. El resultado visible será el personaje apareciendo en distintos puntos de la pantalla.

2. **Interpolado** *(Botón Interpolate)*: Se interpola el movimiento entre paquetes de red. El resultado en pantalla es un movimiento fluido con respecto al anterior. Aparece el *"fantasma"* del personaje indicando la posición marcada en el paquete.

3. **Predictivo** *(Botón Predict)***:** Se predice el comportamiento. Visualmente es muy similar al anterior, pero con diferencias respecto a la relación entre el personaje y su *fantasma*.

Remarcar que el uso del botón ***Slow Motion*** dentro de esta implementación provocará un comportamiento distinto según se utilice en el cliente o en el servidor. Si se usa en un cliente, aparentemente no se verá efecto, sin embargo, si es el servidor el que ralentiza el tiempo, los dos clientes se verán afectados por este cambio.



Pantalla de conexión en los modos de red

Network: Modelo P2P

El botón ***"Non Authoritative Server"*** nos lleva a esta implementación, que permite a dos jugadores jugar una partida competitiva a través de la red.

Para ello, al acceder a dicha implementación, podrán tanto crear un nuevo servidor ***(Botón: "Create Server")***, como unirse a una partida ya existente ***(Botón: "Find Servers")***. El servidor será en este caso a su vez cliente de la partida. Por tanto, se necesitan **dos instancias** de *Beat Fighters* en ejecución: los dos clientes. Ambas instancias pueden estar en distintos equipos.

El segundo cliente podrá conectarse a través de esta pantalla, siguiendo el mismo proceso que en el modelo cliente-servidor. Cuando haya dos jugadores, la partida se iniciará automáticamente.

El uso del botón de ***Slow Motion*** en cualquiera de los clientes provocará, en el mismo cliente, una ralentización del tiempo como sería de esperar. Sin embargo, en el otro cliente, el avatar que representa al jugador rival empezará a dar “saltos”, debido a las constantes correcciones de la posición que se verá obligado a hacer.

ANEXO 2: Glosario

A*

Algoritmo de Inteligencia Artificial utilizado sobre todo en el problema de búsqueda de caminos.

Árbol de Comportamiento

Técnica para el desarrollo del módulo de toma de decisiones en Inteligencia Artificial. Similar a las máquinas de estado, se caracteriza por el uso de tareas y la eliminación de las transiciones explícitas.

ARPANET

Primera red de ordenadores interconectados y precursora del actual Internet.

Cliente (Red)

Cada uno de los jugadores en una partida multijugador.

Cliente-Servidor

Modelo para el desarrollo de aplicaciones para la red, caracterizada por que todos los clientes se conectan a un servidor, y no entre ellos.

Collider (Componente)

Componente en Unity3D que permite detectar colisiones.

Componente

Elemento, como por ejemplo un script, que aporta una nueva funcionalidad a una entidad.

Dead Reckoning

Técnica utilizada en la programación de juegos en red para el cálculo de una nueva posición, a partir de la posición actual y su velocidad.

Estado del Juego (Red)

Es el estado del mundo virtual en un instante de tiempo concreto.

Feedback

Retroalimentación, reacción, o respuesta ante una cierta comunicación.

Frame

Fotograma.

GameObject

Cada una de las entidades de juego en Unity3D. Son contenedores de componentes, que le dan las distintas funcionalidades que necesita. Todo GameObject tiene siempre el componente transform.

GUI

Graphic User Interface o Interfaz Gráfica de Usuario.

Inteligencia Artificial

Área multidisciplinaria que estudia la creación y diseño de entidades capaces de razonar por sí mismas utilizando como paradigma la inteligencia humana.

Latencia

En juegos en red, es el tiempo que tarda en llegar la información a los jugadores.

Línea de Visión (Inteligencia Artificial)

Algoritmo de inteligencia artificial que dota de visión al personaje.

Lobby

Literalmente, "sala de espera". Lugar donde se reúnen los jugadores de juegos en red antes de empezar a jugar.

Mapa de Influencia

Técnica para inteligencia artificial en videojuegos que discretiza el mundo en celdas, asignando a cada celda un valor según un cierto criterio.

Máquina de Estados

Es un modelo computacional que realiza cómputos en forma automática sobre una entrada para producir una salida.

MasterServer

En Unity, plataforma en la que es posible registrar partidas de forma pública para que sean accesibles a través de la red.

MatchMaking

Conjunto de algoritmos para juegos en red que realizan la búsqueda del conjunto de jugadores más idóneo para una partida multijugador.

MMORPG

Massively multiplayer online role-playing game o juego de rol multijugador masivo online.

MonoBehaviour

Clase principal en Unity de la que heredan todos los componentes.

Motor de Videojuegos

Software que busca simplificar el desarrollo de videojuegos ofreciendo al desarrollador soluciones ya construidas para diversos problemas.

Mundo Virtual (Red)

Espacio común en el que conviven todos los jugadores de una partida en red. Contiene a los propios jugadores, escenarios, reglas de juego, etc.

NetworkView (Componente)

Componente en Unity3D que permite la sincronización de un GameObject a través de la red.

NPC

Non Player Character o personaje no jugador.

P2P

Modelo para el desarrollo de aplicaciones para la red, caracterizada por que todos los clientes se comunican entre ellos sin la existencia de un servidor.

Patrón Decorador

Patrón de diseño para la Ingeniería del Software que permite añadir de forma dinámica funcionalidad a un objecto.

Patrón Singleton

Patrón de diseño para la Ingeniería del Software está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto (Instancia única).

Percepción (Inteligencia Artificial)

Capacidad de una entidad de extraer información de su entorno.

Pizarra (Árboles de Comportamiento)

En árboles de comportamiento, las pizarras se utilizan para compartir datos entre distintas tareas.

Prefab

En Unity3D, entidad preconstruida que puede ser reutilizada.

Ray Casting

Técnica consistente en lanzar rayos físicos que colisionan con el componente físico de los objetos y devuelve información acerca de la colisión.

Rigidbody (Componente)

Componente en Unity3D que permite aplicar fuerzas sobre un GameObject.

RPC

Remote Procedure Call o llamada a procedimiento remoto.

Servidor (Red)

En juegos orientados a red, es la unidad central de procesamiento donde se ejecuta el juego.

Steering Behaviours

Técnica para la inteligencia artificial que busca crear movimiento realista a partir de estrategias sencillas.

Tablón de Anuncios (Red)

Aplicación que surgió en los primeros años de internet, que permite a los usuarios intercambiar datos, mensajes y otra información en forma de anuncios.

TCP

Protocolo de la capa de transporte caracterizado por su alta fiabilidad y por ser orientado a conexión.

Time Delay

Técnica de desarrollo de juegos en red utilizada para corregir el problema de la latencia.

Time Wrap

Técnica de desarrollo de juegos en red utilizada para corregir el problema de la latencia.

Transform (Componente)

Componente en Unity3D que dota a la entidad de una posición, rotación y escala, junto a la capacidad de modificar dichos atributos.

Trigger

Literalmente disparador. Entidad que normalmente no tiene componente gráfico que, al ser atravesado por otro objeto, puede poner en marcha una serie de mecanismos previamente implementados.

UDP

Protocolo de la capa de transporte caracterizado por su alta velocidad y por no ser orientado a conexión.

Unity3D

Motor para el desarrollo de videojuegos desarrollado por *Unity Technologie.*

Vertical Slice

Hito o entrega que representa un fragmento del juego de manera muy cercana a cómo será cuando el juego esté terminado, en todas sus capas.

ANEXO 3: Bibliografía

[1] Rusel DeMaria y Johnny Lee Wilson. ***High Score!***

[2] Jesse Schell. ***The Art of Game Design.***

[3] M Tim Jones. ***Artificial Intelligence. A Systems Approach.***

[4] Ian Millington. John Funge. ***Artificial Intelligences for Games Second Edition.***

[5] Aung Sithu Kyaw, Clifford Peters,Thet Naing Swee. ***Unity 4.x Game AI Programming.***

[6] Alex J. Champandard. ***Top 10 Most Influential AI Games.*** http://aigamedev.com/open/highlights/top-ai-games/

[7] Alan R. Stagner. ***Unity Multiplayer Games.***

[8] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, Stephen Wolff. ***Brief History of the Internet.*** *http://www.internetsociety.org/internet/what-internet/history-internet/brief-history-internet#Origins*

[9] *Yahn W. Bernier*. ***Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization.***

[10] *Juan Antonio Recio García*. ***Apuntes sobre Redes para el Máster en Diseño y Desarrollo de Videojuegos de la Universidad Complutense de Madrid.***