UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Mech Battle. Videojuego de estrategia en Unity3D empleando un API de FSM en C#

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Abel Valero Jiménez

Tutor: Dr. D. Ramón Pascual Mollá Vayá

Curso 2015-2016

Resumen

El objetivo del presente trabajo es realizar un juego en 2d de estrategia en el que todas las acciones de los jugadores se escojan previamente y se ejecuten en orden secuencial a su elección simultáneamente. En él se hará uso de una API de gestión de Inteligencia Artificial basada en las Máquinas de Estados Finitos para decidir las acciones de los enemigos. El trabajo se realizará utilizando el motor de videojuegos Unity y el entorno de desarrollo MonoDevelop.

El proyecto será más bien una Alpha en la que se podrán ver las mecánicas básicas del juego, la plataforma que se ha decidido para su lanzamiento será Android.

El API que usaremos ha sido creada por José Alapont Lujá como trabajo final de máster. Esta ha sido desarrollada en C#. Esta API permite crear máquinas de estados finitos y controlarlas de una manera sencilla.

Palabras clave: Video juego, Máquina de estados finitos, Inteligencia Artificial, Unity, C#.

Abstract

The aim of this work is to make a 2D game strategy in which all actions of the players are chosen in advance and executed in sequential order of your choice simultaneously. In it I am using an Artificial Intelligence API management based on Finite State Machines to decide the actions of the enemies, the work is powered using Unity game engine and development environment MonoDevelop.

The project will be rather an Alpha version in which you can see the basic mechanics of the game and the platform decided to start the launch of the project will be Android.

The API we use has been created by Jose Alapont Luja used as his end of master project. This work has been developed in C # and this API allows us to create finite state machines and control them in a simple way.

Keywords: Video game, Finite-state machine, Artificial Intelligence, Unity, C#.

Tabla de contenidos

1. Introducción 8

1.1. Motivación 8

1.2. Objetivos 9

1.3. Metodología 10

1.4. Estructura de la obra 10

1.5. Convenciones 11

2. Estado del arte 11

2.1. Estado del arte 11

2.2. Motores de juegos 12

2.2.1. Elección de motor 14

2.2.2. Plugins y librerías 15

2.2.3. Herramientas 15

2.3. Inteligencia artificial con *Unity* 17

2.4. Referentes 18

2.5. Propuesta 19

3. Análisis 20

3.1. Análisis *Unity* 20

3.1.1. Conceptos básicos *unity* 22

3.2. Análisis *Render* / *Update* 22

4. Diseño 22

4.1. Estructura ficheros 23

4.2. Control del personaje 24

4.3. Acciones del jugador 24

4.4. Mapa 26

4.4.1. Elementos del mapa 27

4.4.2. Objetos del mapa 28

4.5. Colisiones 29
4.6. Diseño de los enemigos 35
5. Arquitectura 39
5.1. Multiplayer 40
5.2. Inteligencia Artificial 40
5.3. *Pool* de objetos 42
6. Conclusiones 43
6.1. Relación con los estudios cursados 43
6.2. Elección del motor *Unity* 44
6.3. Conclusion uso API 45
6.3.1. Número de líneas 46
6.3.2. Tiempo de la creación del primero FSM 46
6.3.3. Tiempo de la creación de Nº FSMs 47
6.3.4. Mantenibilidad de los FSM 47
6.3.5. Escalabilidad de los FSM 47
6.3.6. Valoración general 47
6.4. Problemas surgidos 48
6.5. Gestor de versiones 48
6.6. Versión 5 de *Unity* 50
7. Trabajos futuros 50
8. Bibliografía 52
9. Agradecimientos 52
10. Anexo GDD 53

1. Introducción

En este preámbulo se comentará brevemente que se espera conseguir través de la realización de este proyecto, tanto a nivel personal como académico. También se comentarán cuáles serán los objetivos principales de la obra y la herramienta con la que se desarrollara la IA. Como conclusión, se realizará una pequeña explicación de cómo está compuesto el proyecto.

1.1. Motivación

Desde que era un niño siempre me han apasionado los video juegos. Tenía claro desde entonces que querría dedicarme a ello, y, a través de este proyecto, tomo esta oportunidad para poder desarrollar un videojuego desde cero para así hacer que otros usuarios experimenten las mismas sensaciones que yo sentía de niño, por así decirlo transmitirles una parte de mí. Esta es la principal razón por la que me embarqué en este camino y la que me ha dado fuerzas para poder terminar este proyecto.

Es también una gran oportunidad para la carrera profesional. La industria de los videojuegos crece cada día; en un solo año, la facturación creció un 31%: de 314 millones de euros a 413 millones de euros. Es decir, más del triple del mejor año en taquilla de la historia del cine español y estos datos no paran de sorprender, se cree que pueden llegar a 1.000 millones de euros en 2018.[1]

Gracias a la aparición de herramientas como *Unity 3d, Unreal Engine*, *CryENGINE* la industria ha podido abaratar los costes, no sólo del motor sino también de la formación de los empleados ya que cada día estos motores están siendo más extendidos, más gente los conoce y se apoya en una gran comunidad. Todo esto facilita su aprendizaje, al tener una gran comunidad se crean muchos más plugins para este tipo de herramientas con lo que acorta el tiempo de desarrollo de nuevos proyectos. Por lo que además de reducir costes a grandes empresas ya bien asentadas en el mercado, facilita la incorporación de nuevas empresas. Se puede tomar como ejemplo, el alto número de juegos *indies* que han salido en los últimos años, incluso grandes compañías han apostado por este tipo de productos ya que su inversión es mucho menor y los beneficios pueden ser equiparables a los de un Triple A.

Se puede afirmar que todo el mundo utiliza dispositivos móviles en algún momento del día, el mercado de video juegos para estos dispositivos ha sido de los que más han crecido en los últimos años. Se cree incluso que pueden ser los sustitutos de las consolas portables. Este tipo de dispositivos también han abierto el mercado a nuevos targets que parecían totalmente inaccesibles, pero hará ya unos años que es raro no ver a una señora de 40 - 50 años jugando al *Candy crash* en el metro.

El poder trabajar para una plataforma es un reto añadido ya que hay muchos aspectos a tener en cuenta: la resolución, la RAM, el almacenamiento y la velocidad de procesamiento. Todos estos factores se ven muy mermados en algunos dispositivos y se debe realizar una programación y un control de las imágenes muy riguroso si se busca que funcione correctamente en estos dispositivos.

[1] Fuente de información:
http://cultura.elpais.com/cultura/2015/07/13/actualidad/1436792560_440927.html

Durante este proyecto se pretende adquirir conocimientos en *Unity* y en C#. Además de experimentar el cómo realizar enemigos controlados por IA ya que es un campo realmente interesante.

Con todo esto, pensé que el poder realizar un video juego sería lo más adecuado para la carrera como desarrollador de videojuegos. Las empresas quieren ver de qué eres capaz, las tiendas online como Google Store son plataformas muy accesibles para subir proyectos y para sacar rendimiento económico. La elaboración de este proyecto será, sin duda, la mejor oportunidad para, por un lado, medir los conocimientos adquiridos durante la carrera poniéndolos a prueba con un video juego en 2D y por otro lado, me da la oportunidad de planificar un proyecto plenamente por mí.

1.2. Objetivos

El principal objetivo de este proyecto es la **realización de un juego en 2D** para **plataformas móviles.** En él se desarrollará el primer nivel del modo historia en el que los jugadores competirán contra la máquina (el propio juego). Las mecánicas principales del juego estarán resueltas en este primer nivel y los jugadores podrán disfrutar en este una experiencia completa de las mecánicas del juego.

Se pretende **diseñar varios comportamientos de los enemigos mediante máquinas de estado** diferentes haciendo uso de la API desarrollada por José Alapont Luján con lo que se podrá evaluar dicha API y si fuese posible proponer mejores funcionalidades o sugerir otras nuevas. Para ello también veremos algunas alternativas para la creación de IA que podemos encontrar en el mercado.

También se intentarán adquirir nuevos conocimientos acerca del diseño de video juegos, el crear un juego desde cero es mucho más difícil de lo que parece, todo el mundo tiene ideas descabelladas y fantásticas aportaciones para los juegos. Pero: ¿son divertidas esas propuestas para el usuario?, ¿Son fáciles de desarrollar?, ¿Qué impacto tienen en el desarrollo? Son cuestiones que un buen diseñador de videojuegos deberá tener claros.

Siempre que nos enfrentamos a nuevos proyectos con preguntas que nunca que nos habíamos planteado antes o problemas que no sabemos resolver, la mente del programador crece. Con este proyecto quiero enfrentarme a nuevos paradigmas y poder así obtener experiencia en los problemas futuros, en definitiva, mejorar como programador.

Con el desarrollo de un proyecto de tal envergadura somos nosotros los que nos enfrentamos a problemas grandes en el que en muchos casos ya están resueltos. Por lo que tenemos que movernos con tal de encontrar herramientas y librerías que nos hagan la vida más fácil. Sin estas los tiempos de desarrollo se dispararían. Con este proyecto pretendo encontrar nuevas herramientas que me ayuden al desarrollo de videojuegos.

Objetivos generales

- Aplicar todos los conocimientos adquiridos durante la carrera.
- Continuar con la formación a nivel personal y profesional dentro del marco de los videojuegos.
- Encontrar nuevas herramientas y librerías que puedan facilitar los desarrollos de otros proyectos futuros.

Objetivos específicos

- Poner a prueba la API realizada por José Alapont Luján.
- Utilizar técnicas de IA en video juegos.
- Hacer un juego propio el cual poder enseñar en la vida laboral.

1.3. Metodología

Notación húngara. Todo el desarrollo se ha seguido con esta metodología de trabajo, gracias a ella se consigue dar aún más notación a las variables ya que C# es un lenguaje no tipado y usando esta notación somos capaces de saber más acerca de esta variable como puede ser el tipo. Al principio cuesta acostumbrarse al uso de esta notación, pero una vez se acostumbra, nos damos cuenta de lo útil que es y la facilidad que ofrece para identificar los tipos y alcances de las variables.

Se vuelve fundamental el **tener una buena definición de los requisitos antes de programar**. Aunque parezca obvio, muchas veces, se cae la tentación de ponerse a programar antes de tener bien determinado la lógica y los requisitos que se deberían tomar. Este inconveniente nos lleva a perder tiempo pensando qué se debería hacer y cómo se debería hacer. Sin todo esto bien resuelto, en la fase de definición y toma de requisitos, la programación suele ir más fluida con lo que se gana en tiempo y calidad de código.

1.4. Estructura de la obra

En el siguiente apartado se hablará sobre la **estructura de la documentación entregada para la obra.** Se destacarán los apartados más importantes y los introduciremos muy brevemente. Podemos empezar por la introducción del trabajo donde por primera vez explicamos qué es lo que vamos hacer durante el proyecto y qué es lo que nos ha llevado a afrontar todas estas horas de trabajo. Y al mismo tiempo, también se menciona qué se espera tras el desarrollo de este proyecto.

Se continúa hablando más en detalle sobre la escena de video juegos actual y un breve paso sobre las herramientas y librerías más actuales que podemos encontrar para el desarrollo de video juegos.

Una vez, habiendo echado un vistazo por toda la industria se expone la propuesta de video juego que se ha llevado a cabo para este proyecto de donde se han sacado las ideas y la elección de herramientas, librerías y lenguajes usada para esta obra.

El proyecto sigue hablando sobre el análisis previo al análisis que se ha llevado a cabo para tener los conocimientos necesarios para realizar el siguiente punto: el diseño.

En el diseño se han propuesto formas de solucionar los problemas vistos en análisis. Tras haber obtenido unas soluciones a los problemas, tan solo nos queda comprobar que todo esto es cierto. Por eso en la fase implementación se habla de cómo estos diseños, previamente analizados, se llevan a cabo.

Para finalizar, contaremos las conclusiones obtenidas y qué elementos podrán desembocar en otros proyectos.

1.5. Convenciones

Durante la obra se han mantenido unas convenciones las cuales son las siguientes:

- Las palabras inglesas, los nombres de compañías y los nombres propios de herramientas que se han empleado durante el trabajo aparecerán en cursiva.

2. Estado del arte

En este punto pretendo dar una un vistazo a la industria del video juego de un punto de vista técnico además de ofrecer varios ejemplos reales para poder tener una idea del impacto que han tenido ciertas herramientas en el desarrollo de video juegos en los últimos años.

2.1. Estado del arte

Hace unos años, cualquier proyecto de video juegos tenía un coste altísimo ya que los tiempos de desarrollo también lo eran. Muchas empresas invertían millones en desarrollar motores de juego propios, por lo que la inversión era desorbitada. Algunas de esas empresas, como puede ser el ejemplo de *Epic Games,* decidieron sacar partido a todo ese esfuerzo y abrir el motor para que otras personas pudiesen hacer uso de él; en este caso estamos hablando de *Unreal Engine*.

No todas las empresas tienen el capital financiero y personal suficiente para invertir en un desarrollo tan costoso. Gracias a la aparición de motores de juego como puede ser *Unreal Engine, Unity*, etc… éste panorama ha cambiado muchísimo. Dado que algunas de herramientas son gratuitas o bien a un coste asequible ya que hace unos años cambiaron los tipos de pago y ahora estos pagos son en porcentaje a lo que se ingresa.

Después de estos hechos, en la actualidad la industria posee un vasto abanico de empresas que contribuyen al sector: estudios independientes con ideas algo revolucionaras (*Indis*), empresas bien asentadas en el sector, incluso gente amateur que ha sido capaz de lanzar un juego en equipo o incluso en solitario, como puede ser el ejemplo de *Banished.*

Visto desde otro punto de vista, para los jugadores la cantidad de juegos que salen en un año es abrumadora, por lo que tienen que comparar y seleccionar muy bien cuáles son los juegos que quieren comprarse. Al introducirse tantas empresas nuevas en el sector, incluso dejando sitio para gente amateur, la calidad de los juegos puede bajar y por desgracia muchas veces no se puede llegar a ver los juegos que desde tu punto de vista podrían ser mejores si se llegase a probarlos. Los video juegos que sí se ven son los procedentes de las empresas que invierten más en publicidad o tienen mejores *partners* comerciales.

En esta industria suele haber tres roles bastante diferenciados en los equipos de desarrollo: Desarrollador, Artista y Diseñador de juegos.

Los Desarrolladores son los encargados de: crear los *scrips*, crear las lógicas e incluso montar la estructura del proyecto.

Los Artistas son los encargados de: realizar los diseños gráficos, es decir, *sprits*, texturas, animaciones, etc...

Los Diseñadores de juego son los encargados de: definir las lógicas y ajustarlas, crear niveles, crear personajes, etc.

Es muy importante conocer esta clasificación y su consecuente modo de dirigirnos a estos roles. Sobre todo, cuando nos dirigimos a Artistas y Diseñadores porque suele llevar a muchas confusiones debido a que lo que se conoce en el resto de sectores como Diseñador aquí se llama Artista para acabar con la ambigüedad del Rol de diseñador de videojuegos.

2.2. Motores de juegos

Dado que hoy en día existen multitud de **motores** para la creación de video juegos, veamos ahora cuáles son los **puntos claves** de estos. Estos puntos serán los más influyentes a la hora de llevar a cabo la toma de decisiones y por tanto acabar escogiendo el motor más apropiado para un desarrollo:

Dimensiones en las que es capaz de trabajar el motor, 2D o 3D. Este punto es uno de los más visuales de cara a los usuarios finales ya que ellos lo tienen muy en cuenta. Algunos motores soportan el trabajar para juegos 2D y 3D simultáneamente como puede ser el caso de *Unity*. Este efecto se consigue *fakeando* el 2D.

Lenguajes de programación. Los motores tienen desarrollado un *Core* y se trabaja alrededor de éste. Habitualmente se definen comportamientos de entidades, esto suele hacerse con lenguajes de scripting como pueden ser *Phyton, JavaScript, C#*, etc... Este es importante ya que puede condicionar el uso de un lenguaje u otro, ya que se puede desconocer, lo que implicaría un periodo de aprendizaje o que este lenguaje no sería acoplable al proyecto.

Exportación a plataformas. No todos los motores son capaces de exportar el proyecto a múltiples plataformas, contra más plataformas soporte el motor se podrá llegar a un público mucho mayor. Aunque se pueda exportar a múltiples plataformas no significa que esté optimizado para ellas. Hoy en día los ordenadores de sobremesa disponen de gran cabida de su disco duro, mucha RAM y una capacidad de proceso abrumadora. Por otra parte, los móviles, aunque han tenido una subida de características brutal, no son equiparables con las de un ordenador de sobremesa. Por lo tanto, si el desarrollo está enfocado a dispositivos móviles, será necesario tener en cuenta las prestaciones que ofrecen y aprovecharlas al máximo.

Precio. En los últimos años el tipo de pago de los motores ha ido evolucionando como la industria del software. Antes con gran pago único, se disponía por completo de la versión de motor que habías adquirido. Esta tendencia ha ido cambiando y se está fomentado el pago por suscripción. Como ejemplo, está *unity*. https://store.unity3d.com/es/products/pricing.

Facilidad de aprendizaje. El uso de un motor de video juegos es una tarea muy complicada y lleva años perfeccionar su uso. La curva de aprendizaje que puede tener un motor es en ocasiones demasiado dura y dado que un proyecto está muy acotado por el factor tiempo, este elemento es importantísimo si es la primera vez que se usa un motor concreto.

Comunidad. Este elemento es de los más importantes en la elección de cualquier herramienta de desarrollo. Cada día cuando nos ponemos a trabajar encontramos un montón de problemas y si la comunidad es grande, es muy posible encontrar soluciones inmediatamente. Además, al tener comunidad se desarrollan *plugins* y herramientas que facilitan el desarrollo. Al mismo tiempo, se organizan charlas y eventos en los que se pueden aumentar los conocimientos.

Ahora que ya se han visto los puntos claves que se tienen en cuenta a la hora de la utilización de un motor u otro, se pasa a enumerar un **listado de los motores** más populares y una pequeña descripción de cada uno de ellos:

Unreal Engine 4[2]: Es un motor desarrollado por *Epic Games*, es gratuito si se vende más de 3.000 dólares americanos (USD) y si se llega a esa cifra se paga un 5% de *royalty*. Posee una interfaz gráfica muy potente en la que se puede destacar la herramienta de *Blueprint* que permite a los usuarios "programar de forma gráfica"; o sea, la clásica función de arrastrar y soltar. Es destacado por la calidad gráfica que se puede sacar a este motor. El lenguaje que utiliza es C++ y se puede exportar a múltiples plataformas en las que se incluyen Android e IOs.

Unity 4 y 5[3]: Pertenece a la compañía *Unity* Technologies, es sin duda el motor que más comunidad tiene, es perfecto para introducirse en el mundo de los videojuegos ya que es muy fácil de utilizar en comparación con el resto. Al tener tanta comunidad, se han creado muchos tutoriales muy completos sobre esta herramienta. Cuenta con un *marketplace* con un elevado número de utilidades gratuitas. Lo mejor de todo esto es que *unity* cuenta con una versión totalmente gratuita. Este motor también soporta exportaciones a varias plataformas en las que también se incluyen Android e IOs. Con él se podrá programar en C#, JavaScript y un lenguaje propio llamado UnityScript.

[2] Web Site: https://www.unrealengine.com/what-is-unreal-engine-4

[3] Web Site: https://unity3d.com/es

Cryengine[4]: Creado por *Crytek* para una demostración de *Nvidia*, vieron que tenía tanto potencial y han seguido evolucionando hasta la versión actual V. El cambio más importante de ésta versión no es en el motor sino en el modelo de negocio. Ahora este gran motor se adapta a tus posibilidades ya que puedes pagar por él la cantidad que desees. Además, se puede aportar hasta el 70% de la aportación propia a su fondo de desarrolladores *indis*. Existe material de pago con el que se puede tener acceso a formación y apoyo adicional. Este motor gráficamente es espectacular: se han sacado juegos Triple A con calidades gráficas siendo punteros en su momento. Se podrá programar con el lenguaje *C#* y es exportable a una gran variedad de plataformas. Por desgracia, por el momento, cuenta con una comunidad muy reducida.

También cabe mencionar otros motores como: *GameMaker, Source Engine, Leadwerks Engine* ya que son motores que salvando sus diferencias cada uno tiene algunos puntos fuertes que le han hecho sobrevivir en este mercado tan competitivo como ya se ha podido observar. El proyecto está centrado en sólo tres motores ya que son los que más se adecúan a la idea del proyecto.

2.2.1. Elección de motor

La elección de una herramienta es un tema muy delicado a la hora de iniciar un proyecto de software ya que condiciona todo el desarrollo. Si se toma una mala decisión en la elección de algunas de estas herramientas puedes verte obligado a tener que empezar todo el proyecto desde cero o bien migrar gran parte del proyecto. La elección del motor suele ser una de las decisiones más importantes y por eso se tienen que tener muy claro los pros y contras que te ofrece este motor.

Basándome en mi experiencia y en las características del proyecto la decisión es bastante clara. Para este proyecto se usará *Unity*. De los motores mencionados anteriormente es con este el único motor con el que he experimentado y con el que quiero adquirir todavía más conocimiento. Es, además, el motor más usado por las empresas que conozco del sector. Al ser tan usado, existe mucha documentación y herramientas que complementan este gran motor. Por otra parte, la API de inteligencia artificial que nos ofrece José Alapont está desarrollada en *C#* para *Unity*.

De los tres lenguajes de programación que permite *Unity* nos quedaremos con C#, la comunidad más experimentada de este motor utiliza este lenguaje. Existe una gran cantidad de soluciones a problemas y dudas acerca de este lenguaje para este motor. Los ejemplos que expone el público especializado con *JavaScript* suelen ser de una calidad muy pobre además de generar problemas.

Los contras que se pueden encontrar a este motor son que no es el más completo ni el que proporciona mejores resultados gráficos, pero sobran para un proyecto de estas características.

[4] Web Site: https://www.cryengine.com/

2.2.2. Plugins y librerías

Los *plugins* y librerías proporcionan grandes ventajas a la hora de desarrollar un proyecto. Al haber escogido *Unity* se dispone de una gran cantidad y numerosas herramientas a nuestra disposición dado que es el que más comunidad tiene y fue de los primeros en disponer de una tienda online para todas estas herramientas; además, posee un inmenso catálogo de *assets* para los proyectos.

En esta tienda[5] se encontrará material de pago y gratuito que podremos instalar fácilmente en el proyecto. A continuación, se verán algunos de estos *plugins* o librerías que se han valorado el uso durante la realización del proyecto.

Go-kit [6], es un *plugin* gratuito creado por *Prime31*. Este nos permite generar interpolaciones de diversas propiedades (escala, posición, color...) de una manera muy simple. Tiene una gran variedad de efectos y permite hacer interpolaciones encadenadas. Está bastante desactualizado, pero ya lo había usado en otros proyectos, y sé usarlo perfectamente. Por estos motivos se decidió incluir este *plugin* en el proyecto y no otro.

Si se recomendará un motor de este estilo seria *DOTween*. Además, es uno de los más completos hoy en día, tiene muy buenas críticas y cada día lo utiliza más gente. Cuenta con una versión de pago si la herramienta básica se queda corta, pero la versión básica es bastante potente. Aunque también existen más alternativas como: *iTween* y *LeanTween. Go-kit* cuenta con una documentación algo reducida pero bastante clara, es de código abierto y podemos descargarlo desde https://github.com/prime31/GoKit.

2D Toolkit[7], es un plugin que cuesta 75$ (USD) creado por *Unikron Software*. En la versión de Unity 4.x este *plugin* era casi obligatorio si se quería realizar un juego en 2D. Proporciona muchísimas ventajas a la hora de tratar con *Sprits*. Hoy en día a partir de la versión 5 del motor, no es tan esencial ya que en esta versión se han aportado muchas mejoras para el manejo de *Sprits* pero sigue siendo muy útil. Cuenta con un foro de ayuda con una comunidad propia y bastante activa. La alternativa a este producto es *Orthello 2d Pro* pero no se puede equiparar ya que 2D *Toolkit* es tremendamente completo. Desgraciadamente este *plugin* no se ha incluido en el proyecto. Aunque aporte un gran valor al proyecto tiene un coste demasiado elevado para el proyecto, pero sin duda sería una gran elección el introducirlo en él.

2.2.3. Herramientas

En este punto se van a detallar las herramientas que se van a utilizar para este proyecto, se ha tenido en cuenta que tengan compatibilidad con el motor.

Gestor de versiones, hoy en día es impensable empezar un desarrollo de software sin introducir en este alguna herramienta de este tipo. Este tipo de programas son capaces de llevar una gestión de las versiones del código que se han realizado. Eso se hace guardando esta información en un servidor externo llamados repositorios, los cambios que se tengan en tu local

[5] Web Site: https://www.assetstore.unity3d.com/en/

[6] Web Site: https://github.com/prime31/GoKit

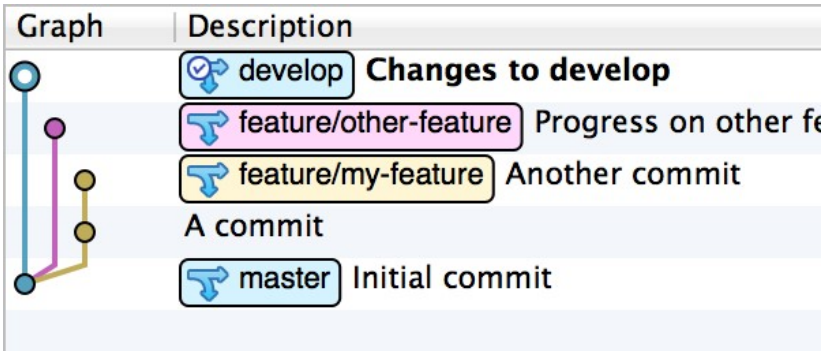
[7] Web Site: http://www.2dtoolkit.com/

son contrastados, con lo que existen el servidor y el propio gestor combina esos ficheros, hace un *merge*, gracias a esto, cosa que proporciona grandes ventajas cuando se colabora en equipo, ya que pueden trabajar varias personas modificando el mismo fichero simultáneamente. En caso de que algún cambio el propio gestor no sepa cómo reaccionar la responsabilidad cae al programador por lo que él debe especificar como deberían ser los cambios que se le aplican a ese fichero.

Cuando se utiliza en grandes proyectos la gestión que se proporciona todavía es más potente porque permite generar ramas desde estados específicos de la aplicación, pudiendo tener en el mismo repositorio versiones bien definidas de un producto específico.

El uso de esta herramienta está muy extendido y existen diversas páginas que son capaz de llevar la gestión de los servidores de este tipo. En este caso se ha utilizado una web llamada *Bitbucket*[8] del grupo *Atlassian*, en los proyectos pequeños la gestión es completamente gratuita.

Source tree, es una herramienta gráfica que aporta poder gestionar el gestor de versiones de una manera más amena, sin necesidad de introducir comandos, además en muchas ocasiones se trabaja con gran cantidad de ramas y es muy difícil llevar ese control en línea de comandos.



Cada uno de esos puntos representa de un *commit* que el estado de la aplicación en un punto determinado. Es una aplicación que proporciona mucha agilidad a la hora de desarrollar.

Mongodevelop es el IDE (entorno de desarrollo) que se ha utilizado, cuando se instala la versión 4 de *Unity* esta viene por defecto con este entorno, es un entorno bastante estándar sin resaltar en nada, pero tiene muy buena integración con el motor, se pueden lanzar procesos en *debug* y *Unity* es capaz de reconocer errores y llevarte a ellos de una manera muy ágil.

En un principio se intentó montar el entorno para que funcionase con de *Microsoft visual studio* en vez de con *Mongodevelop* pero se perdían funcionalidades de integración con *Unity* y por esas razones se desestimó su incorporación.

FireWork es un editor de imágenes que está enfocado a sitios web y aplicaciones, es el editor que sé utilizar y puede valer para hacer una de las primeras versiones del juego, obviamente esta herramienta se queda corta cuando se desean hacer gráficos más detallados, pero son suficientes para esta etapa del proyecto. Se ha utilizado la versión de pruebas CS 6 que nos proporciona todas sus funcionalidades durante 30 días.

[8] Bitbucket web official: https://bitbucket.org/

Personas más en este ámbito utilizan programas como *Illustrator* y *Photoshop.* Son programas que llevan mucha dedicación para aprender a usarlos y se van un poco del propósito de esta obra.

2.3. Inteligencia artificial con *Unity*

El uso que se le da habitualmente a la Inteligencia artificial en los videos juegos es para aportar comportamientos inteligentes a personajes que no son controlados por ningún jugador. Existen dos formas muy extendidas para lograr estos comportamientos con máquinas de estado y con árboles de comportamiento[9].

Para este trabajo nos centraremos en las máquinas de estado. **Estas máquinas se componen por estados y transiciones, conforma una estructura que asigna para cada estado un comportamiento predefinido y según su punto de entrada y las transiciones que va realizando su estado va cambiado con lo que se produce un cambio de comportamiento.**

Las nuevas herramientas que están apareciendo en la actualidad con el objetivo de controlar la Inteligencia artificial de estos personajes no jugadores, se están decantado por la vertiente de árboles de comportamiento. Este tipo de árboles son muy flexibles y ofrecen una gran expresividad.

Se ha decidido que a través del uso de la API de José Alapont, nos permite generar máquinas de estado de diversos tipos (determinista, probabilista, inercial, basada en pilas y concurrente). La elección de está API ha sido por:

- Cuadra perfectamente con los requisitos del proyecto.
- Nos permite gestionar máquinas de estado de manera fácil.
- Nos ofrece la oportunidad de probar un trabajo de final de máster de un compañero.
- No acarrea costes al proyecto.

A pesar de tener ya decidido el uso de la *API* que ofrece José Alapont, a continuación, se verán alternativas que se pueden encontrar en el mercado. Estas herramientas son las más relevantes que se pueden encontrar para el motor de videojuegos que se ha escogido para el proyecto.

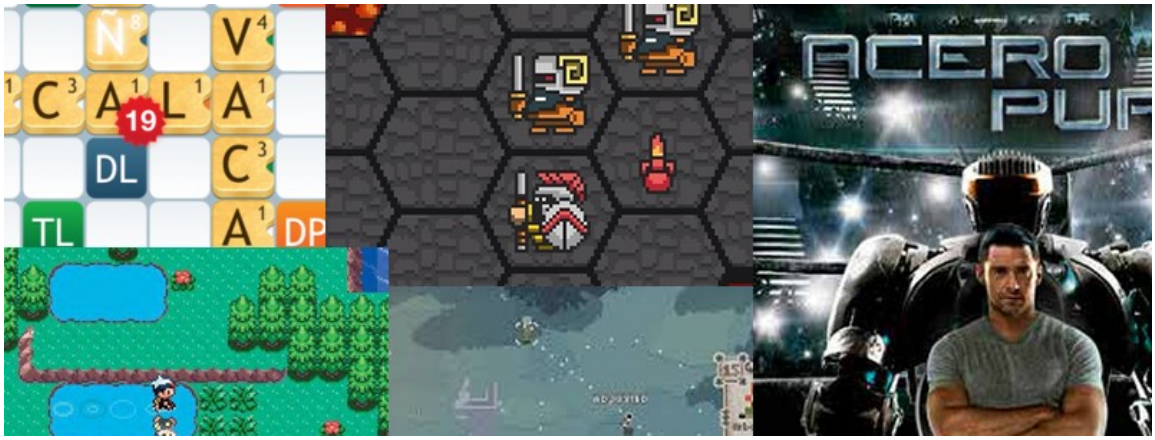
Behaviour Machine. Con esta herramienta se pueden crear de manera fácil máquinas de estado y árboles de comportamiento mediante una interfaz gráfica. Lo más destacado de esta herramienta es que dispone de una documentación bien detallada y videos que explican cómo poder usarla en los proyectos. Se puede adquirir de manera gratuita en la *AssertStore* aunque también cuenta con versiones más avanzadas que valen 50$ (USD) y 250$ (USD).

Behavior Designer. Permite crear árboles de comportamiento con una interfaz gráfica muy intuitiva y completa. Esta interfaz también permite observar y depurar el comportamiento que se ha programado. Tiene muy buena integración con otros *plugin*s de *Unity*. Se puede adquirir por 75$ (USD) en la *AssertStore*.

[9] Behavior Designer - Behavior Trees or Finite State Machines: http://www.opsive.com/assets/BehaviorDesigner/documentation.php?id=49

NodeCanvas. Es una herramienta que permite crear árboles de comportamiento y máquinas de estado. Esta herramienta es la más enfocada a cuestiones de usabilidad ya que va enfocada también a los roles de diseñadores. Tal vez es la que menos flexibilidad ofrece, pero es la más fácil de utilizar. Se puede adquirir por 75$ (USD) en la *AssertStore*.

2.4. Referentes



Hoplite[10], gran juego de estrategia por turnos para móvil en el que turno a turno se tiene que competir contra enemigos hasta limpiar por completo la sala, en él el objetivo es llegar a las salas más profundas donde cada vez atacan más enemigos. Es un juego en el que el personaje va mejorando y ganando habilidades poco a poco. Es un juego con una curva de dificultad muy amena de llevar y tiene un potencial increíble.

Flamberge[11], es un juego en el que primero se deciden las acciones que se desean hacer en un turno para luego ejecutarse simultáneamente con las acciones de los rivales. Es una idea que es brillante y genera una jugabilidad fantástica. Los movimientos son inmensamente flexibles y dispone de una jugabilidad sobresaliente.

Acero puro[12] es una película del año 2011 de ciencia ficción. La película se argumenta en un mundo donde los seres humanos han sido sustituidos por robots de boxeo. Todas estas peleas provocan un gran reclamo para la sociedad. Se pretende crear una esencia similar a la de esta película donde todos sueñan en tener su propio robot de lucha y por supuesto competir en la WRB *(World Robot Boxing)*.

Pokemon[13], sin duda un juego que creó tendencia y que revolucionó el mundo de los juegos que se conocía hasta ese momento. Es un RPG en el que el personaje principal está deseoso de cumplir su sueño que es ser entrenador Pokemon para ganar la liga Pokemon. Es un gran referente para jugadores de los 90. Muchos RPG son muy difíciles de entender porque utilizan mecánicas muy complejas y una tabla de estados demasiado grande. Por el contrario, este juego era simple, aunque también tenía sus trucos para poder sacarle todo el potencial al juego.

[10] https://play.google.com/store/apps/details?id=com.magmafortress.hoplite&hl=es

[11] http://www.indiedb.com/games/flamberge

[12] https://es.wikipedia.org/wiki/Real_Steel

[13] https://www.pokemon.com/es/

Apalabrados[14], Es la versión de móvil del ya famoso juego de mesa *Scrabble*. Tiene una adaptación a la plataforma móvil fantástica. Resuelve de una manera muy buena el sistema por turnos en plataformas móviles, dejando a la espera la partida hasta que el jugador termina la jugada y le pasa el turno a su rival. No ha sido el único juego que ha implementado sistemas de turnos parecidos como pueden ser, *preguntados, atriviate*, etc. Pero fue uno de los primeros y el que he tomado como referente.

2.5. Propuesta

La conclusión que podemos sacar del mundo de los video juegos es que poseen una tendencia al alza. Se conocen, aunque de oídas aplicaciones para móvil que han tenido un éxito arrollador, como puede ser *Candy Crush*, aun así, es un mercado tan grande que quedan muchos nichos que cubrir.

La **propuesta** es hacer un **primer acercamiento a un juego**, es decir, un Alpha **temprana que una vez el juego esté acabado pueda casar en el mercado actual**. Crear un juego casual para un público bastante amplio. Es un juego que por su estructura sería genial disponer de un modo de juego online como el que ofrece *Apalabrados* aunque esta obra solo se ceñirá al uso offline de esta aplicación.

Mech Battel consiste en un enfrentamiento de todos contra todos, donde todos los jugadores luchan entre sí para dominar la arena, y cada jugador podrá elegir unas acciones determinadas: atacar, moverse, esperar, rotar... Es un juego por turnos, donde todos los jugadores tienen un tiempo para elegir un número determinado de acciones, cinco exactamente, y por turno. Una vez todos los jugadores tengan sus acciones elegidas se ejecutarán todas en el orden que ha establecido cada jugador de forma simultánea, en el mismo intervalo de tiempo para todos.

Es un juego donde premia la estrategia, y el azar a partes iguales. Por una parte, se tendrán que predecir los movimientos de los rivales, lo cual no será nada fácil y por otra parte cuenta con un mapa dinámico que da ese toque de aleatoriedad que tanto gusta a los amantes de este género.

Se puede intentar que los enemigos se ataquen entre sí para tan solo rematarlos en los momentos en los que estén más débiles o simplemente destrozarlos con grandes ataques. Lo importante es sobrevivir.

Para crear un modo offline de este juego se va a utilizar el API que proporciona José Alapont para otorgar a los enemigos de inteligencia. De esta manera haremos que sea más divertido el jugar contra los enemigos que nos aparecen en el juego.

Una de las Inteligencias Artificiales que se va a utilizar, va a ser desarrollada por duplicado con la ayuda de la API que José Alapont ofrece y otra sin su ayuda, de esta manera podremos hacer una valoración objetiva del trabajo realizado por él. Se tomarán métricas para ayudarnos a valorar dicha comparación.

Se espera que haciendo uso de la API se reduzca el tiempo de desarrollo y realicemos un código mejor estructurado, que nos permita hacer cambios en las máquinas de estado de una manera más fácil y que dé lugar a menos errores.

[14] http://apalabrados.org/

Para poder evaluar la API que ofrece José Alapont, se pretende evaluar mediante unas métricas con las que podremos dar una valoración de su uso y saber cómo de fácil o difícil es la implementación en un proyecto más cercano a un video juego real. Se van emplear las siguientes métricas:

- Numero de lianas de código utilizadas para realizar la implementación.
- Tiempo de implementación aproximado para el primer FSM.
- Tiempo de implementación aproximado para N FSMs de una complejidad similar.
- Mantenibilidad de los FSM
- Escalabilidad de los FSM.

3. Análisis

En este punto se van a analizar algunos de los factores relevantes para la ejecución de este proyecto, cuando se habla de las problemáticas que tienen los desarrollos de video juegos están más que justificadas por lo que para evitar esos problemas se van a intentar prevenirlas analizando previamente los problemas.

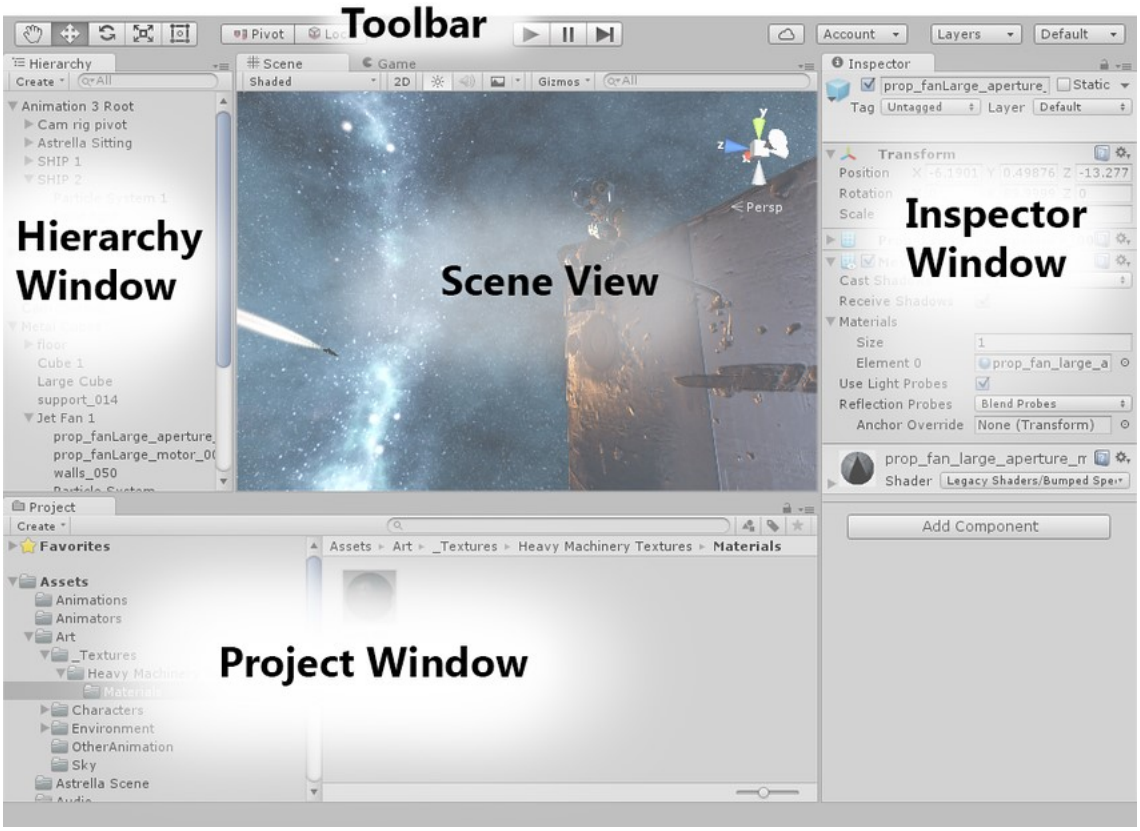
Este tipo de desarrollos tienen peculiaridades que solo se aplican en este campo y se tienen que conocer de primera mano para poder llevar a cabo un proyecto de esta índole.

Por ellos nos ponemos en marca a analizar los puntos clave y algunas herramientas mencionadas anteriormente.

3.1. Análisis *Unity*

El objetivo de este punto es acercarse brevemente al motor *Unity* de esta manera se conocerá como podemos interactuar con él para desarrollar el proyecto.

Primeramente, se va a exponer el interfaz básico que presente una vez se abre una escena:



The Project Window (ventana del proyecto) muestra sus ***assets*** de librería que están disponibles para ser usados. Es como un inspector de documentos enfocado a lo que es la aplicación. Una de las cosas buenas que tiene es que se pueden filtrar por nombre de manera muy rápida.

HirarchyWindows **(**ventana de jerarquía**)** es una representación de texto jerárquico de cada objeto en la escena. Cada elemento en la escena tiene una entrada en la jerarquía, por lo que las dos ventanas están inherentemente vinculadas. La jerarquía revela la estructura de cómo los objetos están agrupados el uno al otro. Encuentre más acerca de la ventana de jerarquía.

Inspecot Window (ventana del inspector) permite visualizar y editar todas las propiedades del objeto actualmente seleccionado. Ya que diferentes objetos tienen diferentes propiedades, el contenido de la ventana del inspector va a variar. En esta ventana se pueden añadir nuevas propiedades a los objetos o bien arrastrándolas y soltándolas sobre este o presionado al botón "*Add Component"*.

La Scene View (vista de escena) permite una navegación visual y editar la escena. La vista puede mostrar una perspectiva 2D o 3D dependiendo en el tipo de proyecto en el que se esté trabajando.

ToolBar (barra de herramientas) proporciona un acceso a las características más esenciales para trabajar. En la izquierda contiene las herramientas básicas para manipular la scene view y los objetos dentro de esta. En el centro están los controles de reproducción, pausa, y pasos. Los botones a la derecha le dan acceso a sus servicios de *Unity Cloud* y la cuenta de *Unity*, seguido por un menú de visibilidad de capas, y finalmente el menú del *layout* del editor (que proporciona algunos diseños alternativos para la ventana del editor, y permite guardar los propios *layouts* personalizados).

3.1.1. Conceptos básicos *unity*

Prefabs, el *prefab* actúa como una plantilla a partir de la cual se pueden crear nuevas instancias del objeto en la escena. Cualquier edición hecha a un *prefab asset* será inmediatamente reflejado en todas las instancias producidas por él, pero, también se pueden anular componentes y ajustes para cada instancia individualmente.

GameObjects, los *GameObjects* son objetos fundamentales en Unity que representan personajes, props, y el escenario. Estos no logran nada por sí mismos, pero funcionan como contenedoras para *Components*, que implementan la verdadera funcionalidad.

Asset, los *asset* con recursos habitualmente externos al motor, aunque el motor es capaz de generar estos recursos. Esto pueden ser imágenes, música, elementos como los *prefabs*, etc...

Scene (escena), una escena se compone por *GameObject*, es decir, cuando se carga una escena se están cargando un conjunto de objetos puestos de una determinada forma y con una configuración determinada para que se visualicen y ejecuten de la manera deseada.

3.2. Análisis *Render* / *Update*

Ese patrón de diseño utilizado en la inmensa mayoría de video juegos es sin lugar a duda el que hace que hace que los video juegos se diferencien tanto en relación a cualquier programa convencional.

Se basa en un bucle que se podría decir que es infinito a menos que se disponga a cerrar la aplicación, este bucle va ejecutando los métodos de *Render* y *Update* de ahí viene el nombre.

La responsabilidad que recae en estos métodos es la siguiente, el método *Render* es el encargado de renderizar las imágenes en la pantalla. Por poner un ejemplo, pinta un cuadrado en la pantalla en las posiciones que tiene establecidas. Por otra parte, el método *Update* tiene la responsabilidad de en base a: acciones, sucesos, etc… toma decisiones para actualizar los valores que están almacenados. Haciendo referencia nuevamente al ejemplo puesto anteriormente del cuadrado, el método *Update* sería el encargado de cambiar las posiciones de este. ¿Entonces qué está pasando?

Dicho de otra manera, el *Render* pinta un cuadrado y el *Update* es el encargado de moverlo por la pantalla.

Por tanto, en el desarrollo que va a realizar es necesario tener muy en cuenta esta manera de programar, ya que la manera de programar es muy diferente a la que se puede estar acostumbrado, se tiene que diferenciar muy claramente la lógica de que debería estar en estos métodos.

4. Diseño

La fase de diseño es fundamental a la hora de la construcción de una herramienta software. Cuando no se diseñan bien las cosas se producen fallos, incoherencias y situaciones no previstas.

En el mundo de los video juegos es muy importante la generación de un documento llamado **GDD** (*game, desing, document*) adjunto en esta obra, este proporciona requisitos y

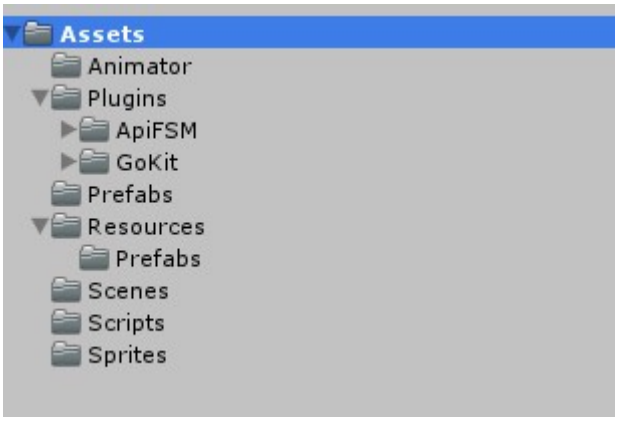
comportamientos relacionados con el juego que resultan de interés a la hora de comenzar fases posteriores a esta. En este documento podremos encontrar entre otras cosas: detalles de concepto, la historia, el arte, los sonidos, la música, etc...

4.1. Estructura ficheros

En cualquier desarrollo software es más que recomendable utilizar una estructura de ficheros lógica con la que tengamos bien diferenciados todos los tipos de ficheros con los que vamos a tratar.

Gracias a mantener una estructura bien definida, se facilita la búsqueda de ficheros. Existen diversos métodos para que tu proyecto esté bien estructurado.

A continuación, se explica la estructura que se ha seguido en este desarrollo. Se ha separado por tipos diferentes.

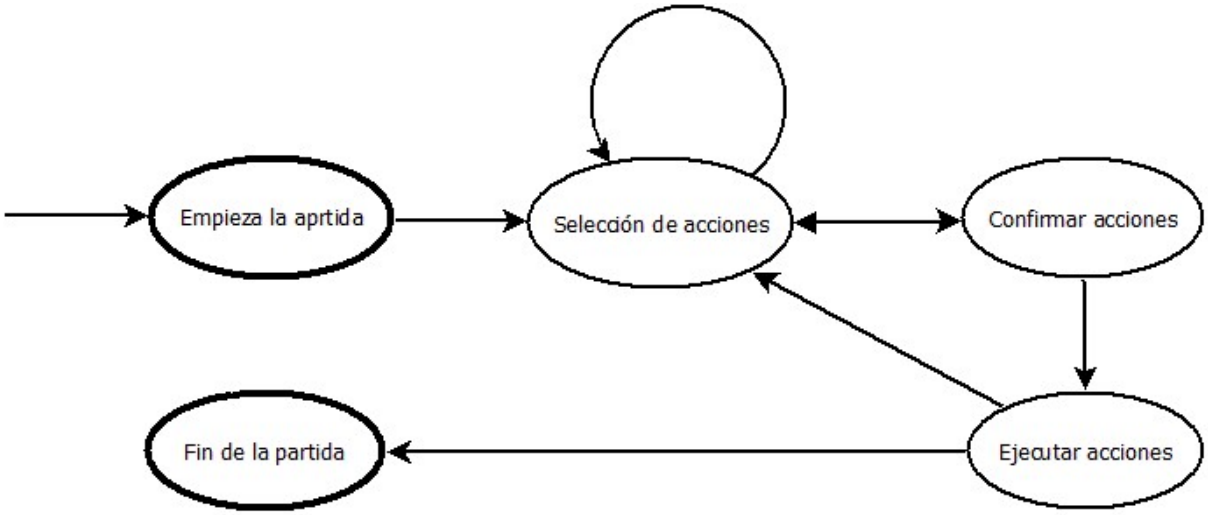


- *Animator* - Contiene todas las *animation* y *animation controller* del juego. Estos son elementos que se utilizan para el movimiento de *sprite*.
- *Plugins* - Contiene todos los plugins usados en el desarrollo (*GoKit* y el API de José Alapont).
- *Prefabs* - Contiene los *Prefabs* del juego.
- *Resources* - Contiene los ficheros *asserts* que se cargan en tiempo de ejecución en el proyecto. Es bueno llevar el control de los objetos que son instanciados en tiempo real, tenerlo separado en carpetas da ese control que se necesita.
- *Scenes* - Contiene todas las escenas que se utilizan en el juego.
- *Scripts* - Contiene todos las *Scripts* del juego.
- *Sprite*s - Contiene todos las *Sprites* del juego.

4.2. Control del personaje

Los Mech son las máquinas que participan en los combates, estas pueden estar controladas por los jugadores o por la IA. Estas máquinas disponen de una cantidad de vida y una potencia de fuego que va aumentado según el nivel del Mech.

Los jugadores controlaran los Mech designándoles acciones a realizar en cada fase de selección de movimiento estas acciones vienen detalladas en el punto de Acciones del juego que se podrán encontrar más abajo. Una vez seleccionadas y confirmadas todas las acciones que el jugador quiere realizar. El juego lanzará todas las acciones síncronamente es decir una de tras de otra hasta finalizar esta fase.



4.3. Acciones del jugador

Llamamos acciones a las cosas que puede realizar un Mech, estas acciones son elegidas por los jugadores y cada uno de los Mech podrá que realizar hasta un máximo de cinco acciones en cada una de sus fases de movimiento, estas acciones están nombradas a continuación, todas las acciones durarán el mismo tiempo. Es decir, el Mech tardará lo mismo en realizar la acción de Moverse que la acción de Rotar hacia la derecha.

Iconos	**Descripción**
	Moverse - El Mech se desplazará una casilla hacia delante, es decir en la dirección a la que apunta su cañón. Si dos o más Mech intentan realizar un movimiento a la misma posición en un momento determinado, el juego elegirá al azar uno de los dos como ganador y será el que ocupe esa posición, en cambio sí se intenta mover a una posición en la que antes había un Mech y este va a dejar libre esa posición, si se podrá realizar este movimiento y ocupar la casilla previamente ocupada.

	Rotar hacia la izquierda/derecha - Esta acción permitirá al Mech rotar 90º en la dirección que elija.
	Esperar - Cuando se elige esta opción el Mech no hará nada. Tras realizar esta acción el Mech será curado una pequeña cantidad. (15 puntos)
	Disparar - Cada Mech es capaz de disparar a una distancia determinada. (Esta acción se podrá potenciar en el Taller)

En este proyecto las acciones son algo muy importante. Estas se han gestionado de una manera para que puedan ser fácilmente escalables, ya que en un futuro pueden haber más acciones que los Mech puedan realizar, por eso se ha creado una clase padre llamada *Action*.

Esta clase tiene la funcionalidad mínima que puede tener una acción. Una acción puede estar bloqueada o estar finalizada. Cuando una acción esta bloqueada esta nunca se realizará y pasará el turno a la siguiente acción.

Para la realización de una acción se han establecido 4 fases. A continuación se explicaran las funciones que se realizan en cada una de estas fases. Las fases siempre se ejecutarán en orden secuencial y para cada uno de los Mech, es decir, la primera fase es *Previus,* el primer Mech ejecuta la fase, después el dos, el tres y el cuarto Mech ahora todos los Mech an acabado con la primera fase, continuarán por la siguiente fase que es *Before…* de esta manera hasta terminar todas las fases.

```
public virtual void Previus(Mech _oMech){
}

public virtual void Before(Mech _oMech){
}

public virtual void Shared(Mech _oMech){
}

public virtual void Run(Mech _oMech){
}
```

Previus, esta es la fase 1. En esta fase se prepara para la realización de una acción. Por ejemplo en la acción de movimiento se activa el predictor de movimiento.

***Before**,* esta es la fase 2. Esta fase es en la que se comprueba que esta acción puede realizarse o no, o dicho de otro modo se comprueba si el Mech está bloqueado, para ello se hacen cálculos y aproximaciones con las que se puede dar por echo que se produce un bloqueo.

Se determina que esta bloqueado por ejemplo: cuando el Mech tiene asignada la acción moverse hacia delante y un obstáculo le impide continuar por su camino.

***Shared**,* esta es la fase 3. Esta fase es en la que comunica el estado de esta acción al resto de acciones. Se puede dar el caso de que una acción sea dependiente de otra. Si una acción A es dependiente de B en el caso de que B quede bloqueada, la acción de A quedará también bloqueada ya que A es dependiente de B. La responsabilidad de bloquear A cae sobre la acción B y se realiza en esta fase.

***Run**,* esta es la fase 4. En esta es en la que se ejecuta la acción en sí. En el caso de disparar donde sale la bala, en el caso de moverse donde se desplaza … etc .

Estar bloqueada no significa que no se ejecute este método, sino que no se realizará como el usuario se espera. Por ejemplo, el usuario tiene que percibir de alguna manera que esa acción de movimiento que él a seleccionando se está realizando correctamente pero algo se lo impide.

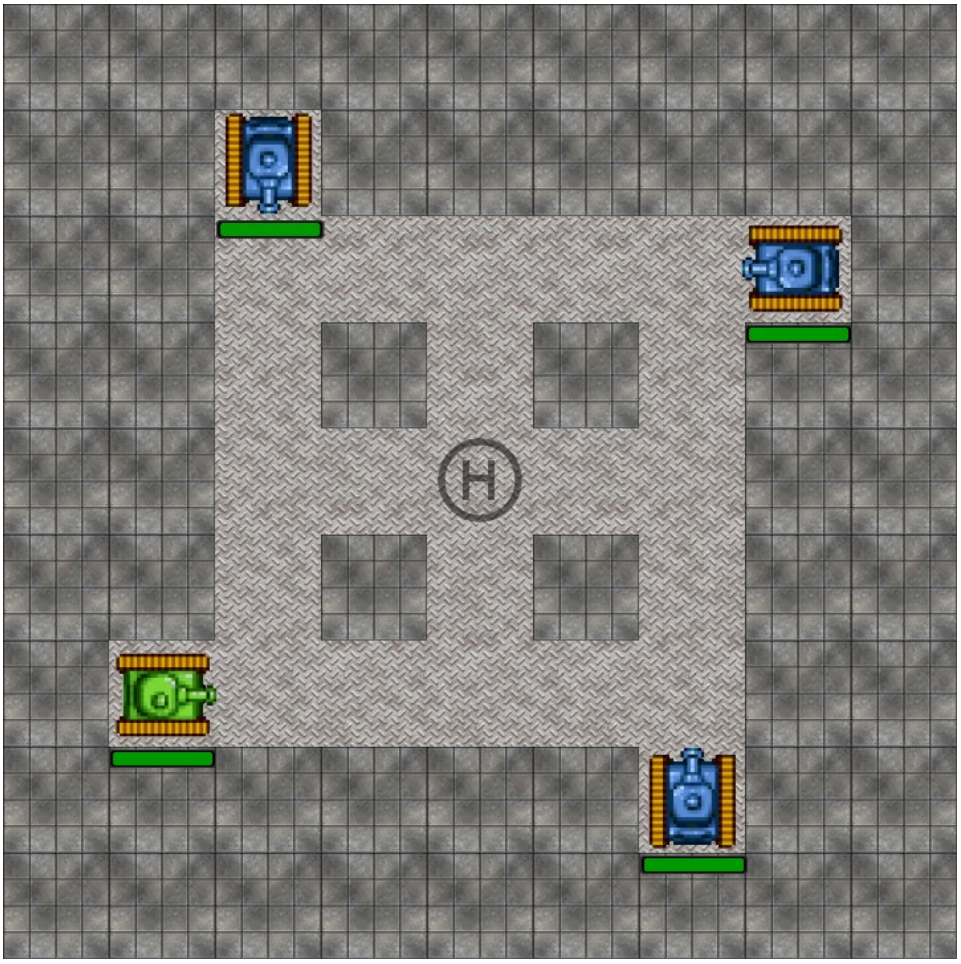
4.4. Mapa

El terreno donde los Mech luchan entre ellos. Se llama arena, esta arena es muy importante ya que condicionará todas las acciones.

Este primer nivel fomenta la igualdad entre todos los Mech, es una arena pequeña pero muy entretenida porque los jugadores más avispados serán los que lleguen antes a conseguir el ítem que aparezca en la zona de *spawn* situada en la zona central del mapa. No será fácil ya que compiten contra otros 3 jugadores que luchan por sobrevivir y apenas se tiene espacio para correr. Este mapa fomenta la lucha entre los Mech y solo los más fuertes son los que saldrán vivos de esta arena.

En las primeras versiones del mapa, era más grande cosa que se hacía el juego muy lento y costaba la interacción con los rivales, por eso se decidió reducir el tamaño. Esto proporcionó una jugabilidad totalmente diferente mucho más rápida y amena.

Gráficos del mapa:



4.4.1. Elementos del mapa

Aunque cualquier objeto colocado en el mapa se podría considerar un elemento del mapa, en este punto trataremos los elmentos que interfieren en la partida que no son ni jugadores, ni muros ni objetos.

Estos elementos son Zona de spaw y Cinta de movimiento.

La zona de spaw está marcada en el mapa con una H en ella apareceran objetos que pueden ser recogidos por los jugadores. Cada objeto aportará unas caracteristicas al Mech que lo recoja.

Esta zona sera capaz de generar un objeto si no existe uno ya en el juego, este objeto será escogido aletoriamente por la máquina y el usuario será el que decida si se arriesga a cojerlo o prefiere que otros luchen por él.

En la fase de movimiento tras ejecutarse una ronda de acciones, existe la posibilidad de que un objeto se cree en esta posición.

La cinta de movimiento esta marcada en el mapa con una flecha hacia una dirección. Esta flecha indica que cuando se termina de ejecutar una acción los Mech que se encuentran en dicha cinta son movidos por esta hacia la direccion indicada. En el caso de de que el Mech necesite ser girado hacia la dirección que señala dicha flecha sera girado y después movido a esa dirección.

Estos movimientos realizados por la cinta se realizan en una fase de movimiento adicional que sucede tras la finalización cada una de las acciones que son realizadas por los Mech, se podría decir que se trata de una post fase.

4.4.2. Objetos del mapa

Los objetos del mapa pueden ser recogidos pro cuaquier Mech que este vive en la arena, estos objetos les proporcionas ciertas aptitudes beneficionas que les ayudaran para ser los campeones en dicha batalla.

Iconos	Descripción
	Herramienta – Esta herramienta cura al Mech 40 puntos de salud.
	Escudo – Este escudo bloquea el siguiente ataque recibido por el enemigo, dando igual el número total de puntos de vida que ese ataque pudiera llegar a restarte.

Estos objetos se usan en el momento de ser cogidos, se pasa por encima de ellos para recogerlos, y su uso es imediato.

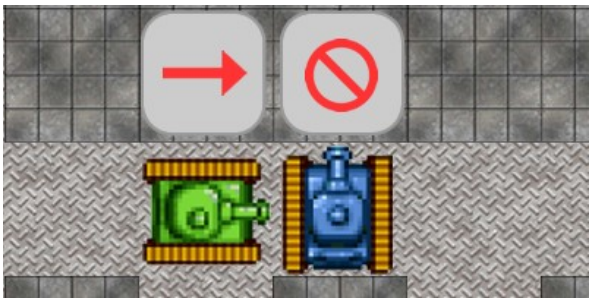
4.5. Colisiones

Una de las cosas más complicadas de este proyecto es la resolución de colisiones. Llamaremos colisión cuando un Mech no podrá realizar una acción de movimiento por estar bloqueado por algo que no puede atravesar como puede ser un muro u otro Mech. *Unity* nos proporciona un sistema colisiones muy avanzado y de una alta calidad, pero para este juego un tanto especial a la hora de los movimientos, este sistema no funcionaba, ya que el sistema que nos proporciona *Unity* funciona con physicas de manera que teniendo un objeto se aplica una fuerza determinada hacia un punto y eso provoca el movimiento, en este caso no se puede hacer de esa manera porque se necesita control total sobre cualquier movimiento o incluso anticiparnos a él, por lo que se utilizan interpolaciones de movimiento con la librería *GoKit.* Este proyecto requiere total control sobre la manera de tratar estos sucesos y gracias a la forma que se ha realizado el sistema de acciones podemos resolver este sistema.

Para esto nos hace falta conocer otro factor muy importante en la resolución del problema de las colisiones, estamos hablando del *predictor*. El *predictor* es la marca invisible para el usuario que se pone en un punto donde se desea mover un Mech. De esta manera podremos predecir todos los movimientos que se van a realizar durante una acción, el predictor se activa en la fase *Previus*. Cuando dos o más Mech intentan ir a la misma posición, se comparará su fuerza con un valor aleatorio; se volverá a calcular el valor aleatorio y el Mech con más valor resultante en esta operación será el que ocupe esta posición.

A continuación, se explica con más detalle que papel hacen las fases de las acciones durante la acción de movimiento para que las colisiones se resuelvan correctamente. En los ejemplos siguientes llamaremos Mech A a el tanque de color verde y Mech B a el tanque de color azul. Pegado a cada uno de los Mech aparece la acción que va a realizar.

Ejemplo 1:



Fase *After*.

Mech A: Activa su *predictor*.

Mech B: No hace nada ya que no realiza una acción de movimiento.

Fase *Before*.

Mech A: Detecta que tiene un Mech en la posición donde deseaba ir y que su acción no es de movimiento, sabe que tendrá que bloquearse.

Mech B: No hace nada ya que no realiza una acción de movimiento.

Fase *Shared*.

Mech A: Se bloquea y comunica a las acciones dependientes de esta que sea ha bloqueado en este caso no tiene ninguna dependencia.

Mech B: No hace nada ya que no realiza una acción de movimiento.

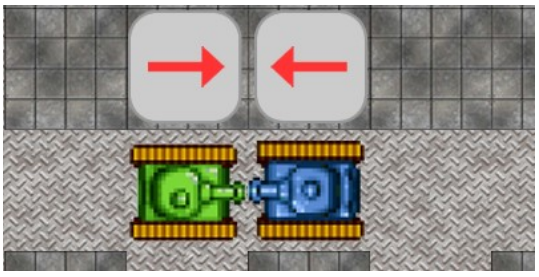
Fase *Run*.

Mech A: Se ejecuta el movimiento como bloqueado.

Mech B: Se ejecuta el esperar.

Como se puede ver en este ejemplo el Mech A queda bloqueado ya que el Mech B le impide el paso. Se comprueba que con el sistema de acciones presentado anteriormente es capaz de resolver este ejemplo.

Ejemplo 2:



Fase *After*.

> Mech A y B: Activa su *predictor*.

Fase *Before*.

> Mech A y B: Detecta que tiene un Mech en la posición donde deseaba ir y que su acción es un movimiento por lo que puede existe una posible colisión. Pide al Mech el vector de movimiento del Mech que ha detectado y lo compara con el suyo propio, al ser un vector opuesto detecta que existe una colisión, ahora sabe que tendrá que bloquearse.

Fase *Shared*.

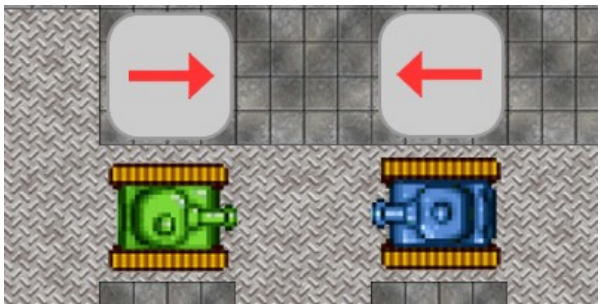
> Mech A y B: Se bloquea y comunica a las acciones dependientes de esta que sea ha bloqueado. En este caso no tiene ninguna dependencia.

Fase *Run*.

> Mech A y B: Se ejecuta el movimiento como bloqueado.

Como se puede ver en este ejemplo el Mech A y B quedan bloqueados ya que intentan ir a la dirección donde está el otro y tropiezan uno con el otro. Se comprueba que con el sistema de acciones presentado anteriormente es capaz de resolver este ejemplo.

Ejemplo 3: Se supone que para este ejemplo la fuerza calculada del Mech A es de 70 por otra parte la del Mech B tiene una fuerza de 30.



Fase *After*.

Mech A y B: Activa su *predictor*.

Fase *Before*.

Mech A y B: Se detecta que un Mech quiere ir a la posición donde este quiere ir. Se realiza una comparación de fuerza y el Mech B sabe que se tiene que bloquear ya que es el que menos fuerza tiene de los dos.

Fase *Shared*.

Mech A: No ha detectado ningún bloqueo.

Mech B: Se bloquea y comunica a las acciones dependientes de esta que sea ha bloqueado. En este caso no tiene ninguna dependencia.

Fase *Run*.

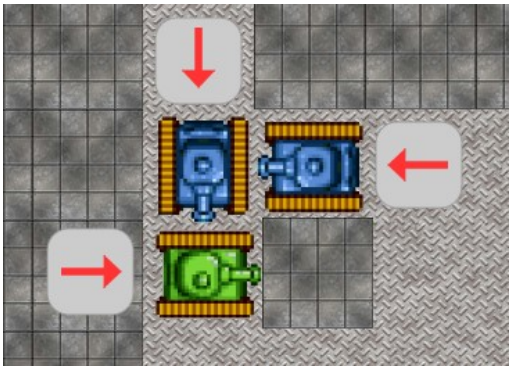
Mech A: Se ejecuta el movimiento con normalidad.

Mech B: Se ejecuta el movimiento como bloqueado.

Como se puede ver en este ejemplo el Mech A es el ganador de la "batalla" y puede moverse al lugar donde él quería. Por otra parte, el Mech B no consigue moverse y queda bloqueado. Se comprueba que con el sistema de acciones presentado anteriormente es capaz de resolver este ejemplo.

Ejemplo 4:

En este ejemplo el Mech B será el tanque azul que intenta ir hacia abajo y el Mech C es el tanque azul que intenta ir hacia la izquierda.



Fase *After*.

Mech A, B y C: Activa su *predictor*.

Fase *Before*.

Mech A: Detecta que tiene una pared delante y es consciente que tendrá que bloquearse.

Mech B y C: Detecta que tiene un Mech en la posición donde deseaba ir y que su acción es un movimiento por lo que puede existe una posible colisión. Pide al Mech el vector de movimiento del Mech que ha detectado y lo compara con el suyo propio, al no ser un vector opuesto desconoce si tiene que bloqueare o no por lo que se suscribe al bloqueo del Mech que ha detectado.

Fase *Shared*.

Mech A: Se bloquea y comunica a las acciones dependientes de esta que sea ha bloqueado en este caso se producirá un efecto en cascada bloqueando B y C.

Mech B y C: Ya están bloqueados.

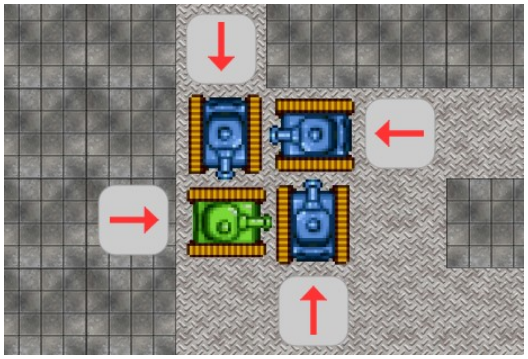
Fase *Run*.

Mech A, B y C: Se ejecuta el movimiento como bloqueado.

Como se puede ver en este ejemplo el Mech A queda bloqueado por el muro y el Mech B y C quedan bloqueados porque dependían C de B y B de A por lo que al bloquearse A se produce un bloqueo en cascada. Se comprueba que con el sistema de acciones presentado anteriormente es capaz de resolver este ejemplo.

Ejemplo 5:

En este ejemplo el Mech B será el tanque azul que intenta ir hacia abajo, el Mech C es el tanque azul que intenta ir hacia la izquierda y el Mech E el tanque azul que intenta ir hacia arriba.



Fase After.

Mech A, B, C y E: Activa su *predictor*.

Fase *Before.*

Mech A, B, C y E: Detecta que tiene un Mech en la posición donde deseaba ir y que su acción es un movimiento por lo que puede existe una posible colisión pide al Mech el vector de movimiento del Mech que ha detectado y lo compara con el suyo propio. Al no ser un vector opuesto desconoce si tiene que bloqueare o no por lo que se suscribe al bloqueo del Mech que ha detectado.

Fase *Shared.*

Mech A, B, C y E: No hacen nada ya que ninguno ha detectado bloqueo.

Fase *Run.*

Mech A, B y C: Se ejecuta el movimiento con normalidad.

Como se puede ver en este ejemplo el Mech A, B C y E no quedan bloqueados ya que se mueven todos en espiral, todos pueden realizar su movimiento con normalidad. Se comprueba que con el sistema de acciones presentado anteriormente es capaz de resolver este ejemplo.

4.6. Diseño de los enemigos

A la hora de implementar a los enemigos se les ha implementado un FSM el cual recorre una serie de estado que hará que el comportamiento del enemigo varíe, gracias a unos "sensores" el enemigo es capaz de percibir toda esa información y actuar en consecuencia. Cuando sucede un evento que se considera importante para el desarrollo de un FSM este lo recibe y es capaz de tomar una decisión con esa nueva información.

A continuación, se explicarán los FSM que se tienen en cuenta para el desarrollo de estos. Para nuestra primera pantalla se han diseñado 3 enemigos que actúan de manera diferente ya que su FSM es diferente a pesar de que tengan estados muy similares las transiciones entre los estados la y escucha de diferentes eventos les proporciona cierto comportamiento distintivo para cada uno de los enemigos.

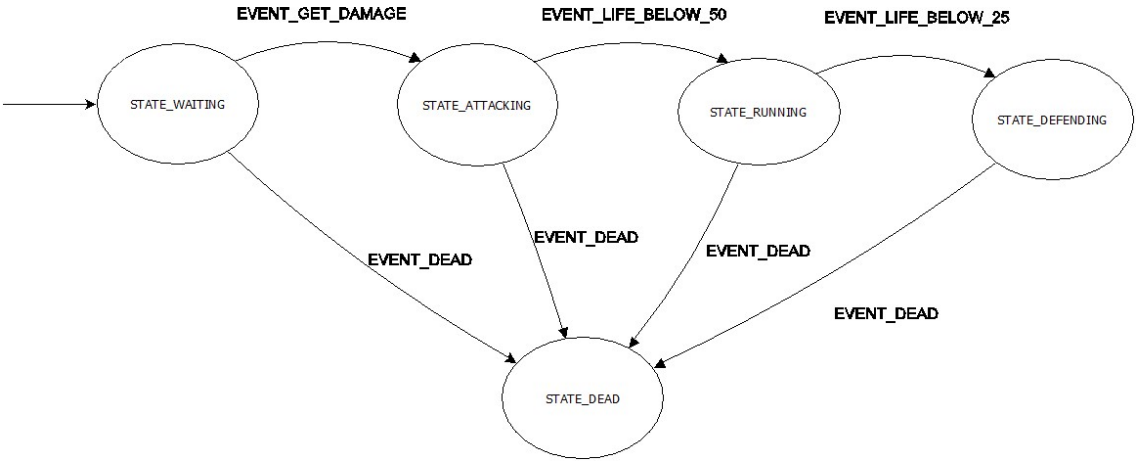
Pero antes de entrar más en profundidad en los comportamientos de los FSM se va a explicar todos los eventos y estados que aparecen en estos ya que tienen el mismo comportamiento para cada uno de estos enemigos.

Comportamiento de los Mech en los estados del FSM

Estado	Descripción del comportamiento.
STATE_WATING	El Mech se quedará parado sin hacer nada
STATE_ATTACKING	El Mech mirara en 3 direcciones, hacia delante, hacia la derecha y hacia la izquierda. Este se encarará a su rival y si dispondrá a disparar.
STATE_RUNNING	El Mech irá hacia delante hacia delante y si tiene un obstáculo delante girará a la izquierda.
STATE_DEFENDING	El Mech se quedará parado sin hacer nada ya que esta opción también cura quince puntos de salud.
STATE_LOOKING_ENEMY	El Mech irá hacia delante hacia delante y si tiene un obstáculo delante girará a la derecha.
STATE_DEAD	El Mech no realiza ninguna acción ya que está totalmente debilitado.

Por otro lado, se puede observar que eventos suceden para que estos se lancen. Estos eventos pueden ser lanzados por cualquier objeto que interactúe con el juego, pero los encargados para lanzar dichos eventos son los que tienen acceso a un Mech ya que el que tiene la lista de eventos que van sucediendo en cada momento.

Evento	Suceso que hace que se lance
EVENT_LIFE_BELOW_50	El Mech recibe daño y está por debajo del 50% de la vida.
EVENT_LIFE_BELOW_25	El Mech recibe daño y está por debajo del 25% de la vida.
EVENT_DEAD	El Mech muere.
EVENT_ENEMY_NOT_FOUND	El Mech no detecta ningún enemigo cerca.
EVENT_ENEMY_FOND	El Mech detecta ningún enemigo cerca.



Este FSM determinista que está compuesto por cinco estados diferentes, los cuales son: *STATE_WATING, STATE_ATTAKING, STATE, RUNNING, STATE DEFENDING* Y *STATE_DEAD*. Además de contar con cuatro eventos que es capaz de detectar los cuales son: *EVENT_GET_DAMAGE, EVENT_LIFE_BELOW_50, EVENT_LIFE_BELOW_25* y *EVENT_DEAD.*

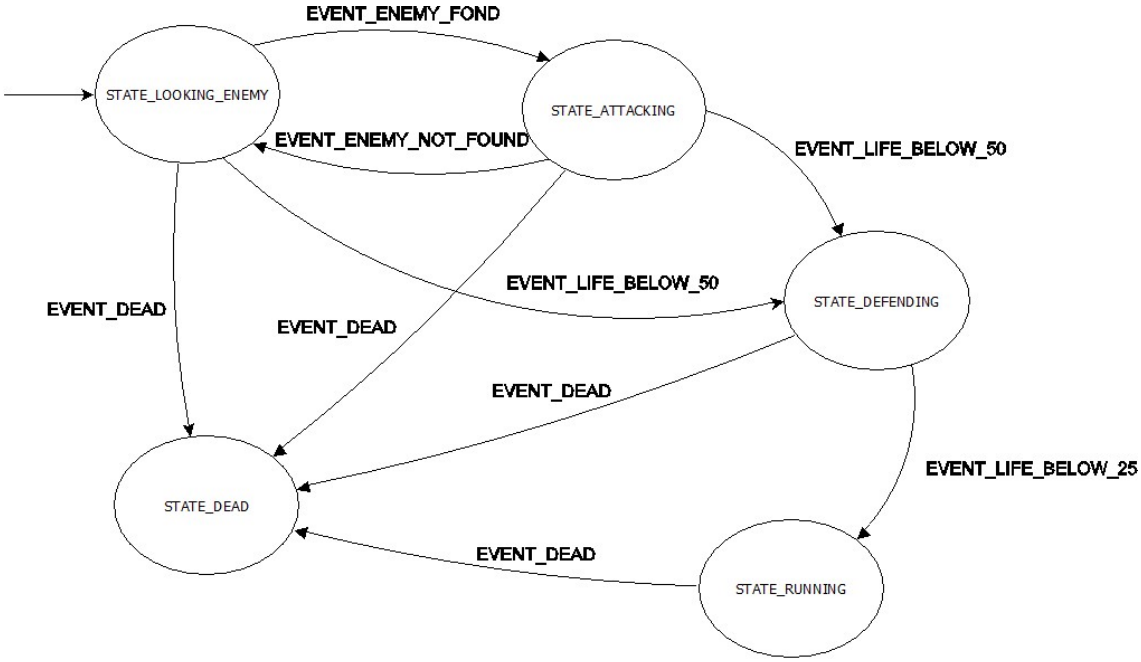
El comportamiento que se busca para este enemigo es que en un principio este parado en medio del campo de batalla hasta que reciba daño. Una vez recibe daño pasará al estado de ataque, cuando este se vea debilitado correrá para evitar ser alcanzado nuevamente. Si a pesar de ello sigue siendo atacado pasara a una posición aún más defensiva.

A pesar de que existen herramientas en el campo de batalla para curar a los Mech debilitados no sea a implementado en los eventos *EVENT_LIFE_OVER_50* ni *EVENT_LIFE_OVER_25* que podrían hacer que del estado *STATE_RUNNING* vuelvan a pasar al estado *STATE_ATTAKING*, para que el jugador principal pueda acabar con este enemigo fácilmente ya que tan solo recibirá daño cuando está en su estado de *STATE_ATTAKING.*

A continuación, se verán dos enemigos que son muy similares salvando algunas pequeñas diferencias.

Algunas de esas pequeñas diferencias son que uno de ellos contiene alguna transición más, pero la más importante es que uno de ellos es un FSM determinista y el otro es un FSM probabilístico.

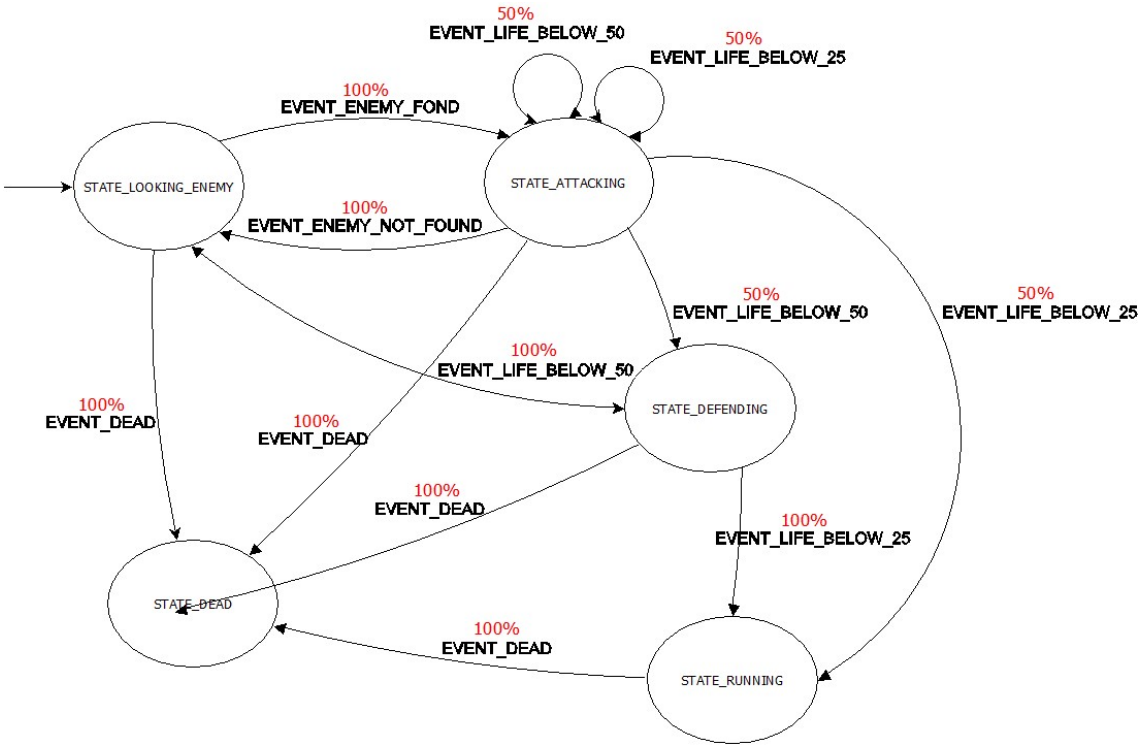
Gracias a que estos FSM son bastante parecidos destacan aún más sus diferencias y podemos compararlas de una manera más clara.



Este FSM determinista que compuesto por cinco estados diferentes que son: *STATE_LOOKING_ENEMY, STATE_ATTAKING, STATE, RUNNING, STATE DEFENDING* y *STATE_DEAD*. Además de contar con cuatro eventos que es capaz de detectar, que son: *EVENT_ENEMY_FOUND, EVENT_ENEMY_NOT_FOUND, EVENT_LIFE_BELOW_50, EVENT_LIFE_BELOW_25* y *EVENT_DEAD.*

El comportamiento que se busca para este enemigo es que un principio busque al jugador recorriendo el mapa, en caso de encontrarlo que se disponga a pelear con él atacándole ferozmente, si este es debilitado pasará a una actitud más defensiva. Si a pesar de todo eso su vida se ve muy afectada, que se disponga a huir de él.

A pesar de que existen herramientas en el campo de batalla para curar a los Mech debilitados, no sea ha implementado en los eventos *EVENT_LIFE_OVER_50* ni *EVENT_LIFE_OVER_25* que podrían hacer que del estado *STATE_RUNNING* vuelvan a pasar al estado *STATE_ATTAKING*, para que el jugador principal pueda acabar con este enemigo fácilmente ya que tan solo recibirá daño cuando está en su estado de *STATE_ATTAKING.*



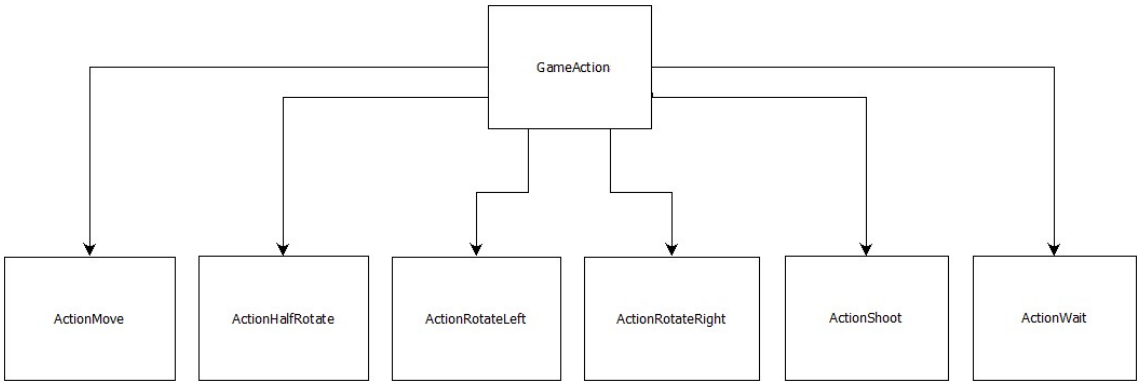
En la imagen superior podemos encontrar un FSM probabilístico, en el que aparecen en rojo para cada una de las transiciones la probabilidad de que, si un evento sucede pase al siguiente estado por dicho camino o permanezca el actual, este FSM está compuesto por de cinco estados diferentes los cuales son: *STATE_LOOKING_ENEMY, STATE_ATTAKING, STATE, RUNNING, STATE DEFENDING Y STATE_DEAD*. Además de contar con cuatro eventos que es capaz de detectar: *EVENT_ENEMY_FOUND, EVENT_ENEMY_NOT_FOUND, EVENT_LIFE_BELOW_50, EVENT_LIFE_BELOW_25* y *EVENT_DEAD.*

El comportamiento que se busca para este enemigo es que un principio busque al jugador recorriendo el mapa, en caso de encontrarlo que se disponga a pelear con él atacándole ferozmente. Si este es debilitado podrá pasar a una actitud más defensiva o bien permanecer atacando. Este suceso podrá pasar cada vez que recibe daño estando por debajo del 50%. Si a pesar de todo eso su vida se ve muy afectada nuevamente por debajo del 25%, si estaba atacando podrá optar por permanecer atacando o prepararse para huir. Por otra parte, si ya se encontraba en una actitud más defensiva se verá obligado a huir.

A pesar de que existen herramientas en el campo de batalla para curar a los Mech debilitados no sea ha implementado en los eventos *EVENT_LIFE_OVER_50* ni *EVENT_LIFE_OVER_2,* que podrían hacer que del estado STATE_RUNNING vuelva a pasar al estado *STATE_ATTAKING*, para que el jugador principal pueda acabar con este enemigo fácilmente ya que tan solo recibirá daño cuando esté en su estado de STATE_ATTAKING.

5. Arquitectura

En este punto destacaremos las claves principales de la arquitectura desarrollada para este proyecto, se ha tenido muy en cuenta la escalabilidad y la creación de pequeños módulos separando muy bien la lógica de algunos objetos. Lo que se ha pretendido hacer desde el principio es preparar el proyecto para futuras ampliaciones. Teniendo esto en cuenta se han tomado decisiones que en principio dificultaban la creación del proyecto, pero se han visto recompensadas a la hora de implementar nuevos objetos o nuevas dinámicas.



Como se muestra en la imagen superior la clase *GameAction* es la padre de todas las acciones y todas las hijas heredan su comportamiento. Esta clase padre contiene las fases de una acción: *Previus, Before, Shared, Run*. Además, cuenta con las propiedades que definen el estado de una acción, si está bloqueado o ya ha finalizado.

La manera de ejecutarse una acción es en el orden mencionado anteriormente cada una de estas fases es ejecutada por cada uno de los Mech. Hasta que todos ellos no finalicen una fase, no empiezan la siguiente. En otras palabras, no puede ocurrir que un Mech se encuentra en la fase *Previus* y otro en la fase *Before.*

La manera con la que se puede garantizar que todos los Mech van pasando de manera síncronizada por cada una de las fases es mediante barreras. Este tipo de herramientas son muy utilizadas cuando se emplean hilos, en este caso ha hecho falta porque *GoKit* lanza hilos de manera invisible para el programador por tanto con esta implementación aseguramos que esto se cumple.

```
for(int i = 0; i < m_oMechs.Length; ++i ) {
    if( m_oMechs[i].active && !m_oMechs[i].IsDead() ) {
        while( !m_oMechs[i].isFinishedAction() ) {
            yield return new WaitForSeconds(1.1f);
        }
    }
}
```

Por poner un ejemplo de la modularización que dispone este proyecto, las acciones están completamente desvinculadas de quien realiza la acción. Esto da una gran flexibilidad ya que desvincular los Mech de las acciones ha ayudado a poder aprovechar todo ese trabajo para usar las mismas acciones para la cinta de movimiento.

De la misma manera se han realizado los objetos consumibles del juego, el escudo y la herramienta. Estos objetos tienen la misma interfaz y aunque parezca que realizan acciones muy diferentes a la hora de la implementación, son acciones que afectan a un Mech pudiendo modificar cualquiera de sus propiedades. Al actuar de esta manera tan homogénea no solo se facilita la comprensión del código, sino que también se abre la puerta a implementar nuevos consumibles. Como bien se ha mencionado antes la escalabilidad se ha tenido muy en cuenta.

5.1. Multiplayer

Muchos de los juegos de hoy en día son *multiplayer*. Este tipo de juegos requieren una arquitectura que sea capaz de admitir este tipo de modos de juego. El modo *multiplayer* no se ha implementado completamente, pero se le ha abierto un camino para que esta opción esté disponible lo antes posible ya que se ha tenido muy en cuenta durante todo el desarrollo la posibilidad de que los otros Mech sean manejados por otro jugador.

De hecho, los Mech que podemos encontrar en la pantalla disponible no están implementados como enemigos, es la IA que se hace pasar por un jugador normal y nos dice los movimientos que haría dicho jugador.

Al haber pensado desde un principio que la IA era un Jugador no un simple enemigo, todo el diseño de los Mech se ha visto influenciado por eso y esto nos ha permitido abrir ese camino al modo *multiplayer.*

5.2. Inteligencia Artificial

El papel que ha tenido la IA en el proyecto ha sido fundamentalmente la capacidad de actuar como otro jugador, no solo como un simple enemigo más. Esto nos proporciona algunas ventajas como dejar el camino abierto a la implementación del modo multijugador, pero complica en gran medida el desarrollo, ya que a nivel de código el Mech que se lleva es exactamente el mismo que el enemigo.

La IA se ha realizado mediante máquinas de estado. Esta tiene un papel fundamental para la obra y ha condicionado mucho el desarrollo no solo por la implementación de las máquinas de estado, también se ha tenido que dotar de cierta ventaja a los Mech controlados por la IA para que sean capaces de “predecir” tus movimientos, para ello en vez de generar un árbol de posibles acciones para N movimientos se ha realizado una arquitectura para que los Mech de la IA sean capaces de decidir en cada turno que acción hacer. ¿Cómo es esto posible?, ¿No se ha dicho que El Mech encontrado por la IA es el mismo que el de cualquier otro jugador?, ¿Por qué no se pueden decidir las acciones en cada turno?

Para un Mech que está controlado por un jugador, pulsa sobre cualquier acción, por ejemplo, moverse hacia delante, lo que estamos haciendo es encolar una “promesa de acción” que en este caso es una función que devolverá la acción moverse hacia delante. Cuando seleccionemos las cuatro acciones restantes se resolverá esa promesa y se ejecutará la acción que seleccionamos, moverse hacia delante.

Ahora bien, cuando la IA confirma sus cinco acciones, a pesar de haberlas confirmado, todavía no sabe qué acciones son las que va a ejecutar. Únicamente ha confirmado que realizara cinco acciones con lo que tenemos su promesa. De esta manera podemos dejar esta decisión para

tomarla en el momento antes de esta acción sea ejecutada por el Mech, gracias a esto tiene la información en tiempo real de lo que ha sucedido en el "paso" anterior para el resto de jugadores.

De esta manera lo que conseguimos es:

- La IA cuenta con una información extra y fiable.
- Nos ahorramos coste computacional para resolver un árbol de 5 niveles de posibles acciones.
- Nos proporciona gran flexibilidad a la hora de desarrollar nuevas acciones. Por ejemplo, podríamos crear un ejemplo que cambiase la acción del resto de jugadores de una manera muy fácil.
- El comportamiento de la IA es exactamente el mismo que de un humano, por lo que se podría sustituir por este ya que la IA también confirma cinco acciones antes de empezar la fase.

```
using UnityEngine;
using System.Collections;

// Clase que actua como una promesa
public class FunAction {

    //Definiicion del tipo de callback
    public delegate GameAction FunctionAction();

    //Atributo en el cual se almacena el callback
    // por el constrctor
    public FunctionAction callbackFct;

    //Guarda el Callback en la construccion de la calse
    public FunAction( FunctionAction _fAction ) {
        callbackFct = _fAction;
    }

    //Ejecuta el callback y lo devuelve
    public GameAction Get( ) {
        return callbackFct();
    }

}
```

En esta foto se puede ver cómo es esto posible con esta pequeña clase. El potencial que se ha podido obtener gracias a esta implementación.

5.3. *Pool* de objetos

Uno de los patrones de diseños utilizados en el proyecto es el de *Pool* de objetos, no es un patrón exclusivo del diseño de video juegos pero suelo usarse en la mayoría ya que aporta grandes beneficios respecto al rendimiento.

Este patrón surge de la necesidad de instanciar una serie de objetos en un momento determinado, en este caso se ha implementado para la gestión de balas, pero existen muchos más ejemplos. Cuando se crea una instancia de un objeto se está consumiendo una cantidad de recursos más elevada de lo normal.

Por lo tanto, para mejorar el rendimiento, instancian al principio del proyecto un número N de objetos para tenerlos en el *Pool*. En otras palabras, en un cajón guardado hasta que esos objetos se precisen. Cuando este objeto se pide al encargado del gestionar el *Pool* un objeto ya instanciado en él, el gestor irá asignado sus objetos conforme se los vayan pidiendo, en nuestro caso de manera cíclica.

```
public class ManagerBullets : MonoBehaviour {

    private const int MAX_BULLETS = 16;
    static private int m_iCurrentBullet = 0;

    static private Bullet[] m_oBullets;

    // Use this for initialization
    void Start () {
        Initialize();
    }

    public void Initialize() {
        m_oBullets = new Bullet[MAX_BULLETS];

        //Crea todas las instancias del poll
        for(int i = 0; i < MAX_BULLETS; ++i){
            //Forma de instanciar un objeto en Unity
            GameObject oBullet = (GameObject)Instantiate(Resources.Load("Prefabs/Bullet"));

            //Lo creamos como hijo de este gameObject para mejorar la
            //visibilidad de la gerarquia de objetos en unity
            oBullet.transform.parent = transform;
            m_oBullets[i] = oBullet.GetComponent<Bullet>();
            m_oBullets[i].Initialize();
        }
        //Este sera el indice del primer objeto devuelvo
        m_iCurrentBullet = 0;
    }

    //Lanza el objeto del pool
    static public void Launch(Vector3 _vPositionOrigin, Vector3 _vAngles, int _iDistance, int _iMechShooterID){
        m_oBullets[m_iCurrentBullet].Launch(_vPositionOrigin, _vAngles, _iDistance, _iMechShooterID);
        //Incrementamos el indice y si llega al maximo vuelve al primer objeto
        m_iCurrentBullet = ++m_iCurrentBullet % MAX_BULLETS;
    }
}
```

Como se muestra en la imagen, esta es la implementación que se ha realizado para este patrón en este proyecto, el uso que se le ha dado ha sido para gestionar las balas que disparan los Mech. Como bien se ha explicado anteriormente es un ejemplo claro de uso de este patrón, incluso suelen exponerlo como ejemplo en muchos tutoriales.

6. Conclusiones

En esta obra se ha llevado acabo el desarrollo del Alpha de un video juego. Un juego en 2d de estrategia, del cual se pueden extraer las mecánicas básicas que dispondrá este juego. La construcción de un juego no deja de ser un desarrollo software y además de contar con la complejidad que conlleva cualquier desarrollo le sumamos una serie de dificultades añadidas que vienen por ser un video juego.

Si no fuese por herramientas como Unity este desarrollo se hubiese alargado en el tiempo además de haber aumentado increíblemente si dificultad, haciendo casi imposible su desarrollo en unos tiempos tan acotados.

6.1. Relación con los estudios cursados

Cuando se habla de los estudios de informática, cualquier asignatura cursada durante la carrera puede ser digna de mencionada en este punto ya que todas nuestras enseñanzas son construidas unas encima de otras como si se tratase de un castillo de naipes.

Por ello solo se mencionarán las más destacadas o en las que en mi opinión son más importantes:

Sincronización y concurrencia: La programación de un video juego tiene algunos aspectos bastante difíciles de implementar respecto a la programación de cualquier otro sistema, ya que cuenta con el patrón *render update* e hilos.

Para poder implementar muchas opciones se ha hecho uso de: eventos, barreras de sincronización, etc… este tipo de herramientas son muy usadas en los sistemas distribuidos.

Asignaturas como **concurrencia de sistemas distribuidos** me han ayudado mucho en estos aspectos.

Matemáticas y física: A la hora del desarrollo de un video juego son fundamentales estas dos disciplinas, para casi cualquier cosa se hace uso de estas disciplinas: movimientos, colisiones, interpolaciones, etc. son algunos ejemplos. Sin lugar a duda la más utilizada en el mundo de los video juegos es la trigonometría y el álgebra ya que cualquier objeto en el escenario está representado por un vector.

Asignaturas como **Álgebra, Análisis Matemático y Fundamentos físicos de la informática** me han ayudado mucho en estos aspectos.

Programación y Desarrollo de software: A la hora de enfrentarte a un video juego creado desde cero no solo te vale con saber programar, sino también se necesitan de conocimientos diseño de software, estructuración de código, arquitectura de software, patrones de diseño.

Cuando se desarrolla un video juego se necesita que las respuestas a las acciones de los jugadores sean muy fluidas, si no se hace de este modo da la sensación de que el juego funciona mal ya que el volumen de información que se tiene que tratar es muy elevado necesitas mecanismos que resuelvan de manera muy eficiente cualquier obstáculo que podamos encontrar por el camino.

Asignaturas como **Estructuras de Datos y Patrones de Diseño** me han ayudado mucho en estos aspectos.

Autómatas y máquinas de estado: una de las partes más importantes de esta obra ha sido la utilización de la API de José Alapont la cual permitía la gestión de máquinas de estado. Aún sin haber sido una parte destacada del trabajo, las máquinas de estados son muy usadas en los desarrollos de video juegos por la necesidad de ejecutar unas acciones determinadas para ciertos estados.

Los conocimientos adquiridos en las asignaturas Teoría de Autómatas y Lenguajes Formales, Inteligencia Artificial e Ingeniería de Sistemas y Automática han sido de gran ayuda.

Asignaturas como **Teoría de Autómatas, Inteligencia Artificial y Lenguajes Formales** me han ayudado mucho en estos aspectos.

Comparación de código: La comparativa realizada al API de José Alapont es un reflejo más de la finalización de un curso ya que tras este ha habido algunas asignaturas que nos han proporcionado herramientas necesarias para dicha evaluación.

Asignaturas como **Calidad y Mantenimiento de software** han ayudado mucho en estos aspectos.

Introducción a los video juegos: Durante el curso académico ha habido dos asignaturas en las que se han tocado más de cerca el mundo de los video juegos que tanto me ha apasionado desde la infancia, estas asignaturas no solo han sido impulso para a la realización de este TFG, sino que también han ayudado a entender como estaban hechos los video juegos, ellas son **Iniciación a la programación de video juegos** y **Arquitecturas y entornos de desarrollo de videoconsolas**.

Sin lugar a duda muchas asignaturas han ayudado de una manera u otra al desarrollo del proyecto. Ya que el conjunto de todas son las que han forjado la base de todos los conocimientos adquiridos, algunas otras han ofrecido una disciplina necesaria para ejecutar un proyecto software de esta envergadura.

6.2. Elección del motor *Unity*

La elección del motor es una decisión muy importante en este tipo de desarrollos y por tanto es bueno reflexionar sobre su elección.

Sin lugar a duda *Unity* le ha aportado un gran valor a este proyecto, gracias a él se ha podido realizar un proyecto de video juegos, el cual personalmente, era uno de los hitos marcados para mí para el proyecto.

He aprendido muchas cosas a tener en cuenta a un desarrollo desde cero con el motor además de que ahora cuento con experiencia demostrable con el enfocado de esta manera mi carrera profesional poco a poco al mundo de los video juegos.

A pesar de haber tenido un gran periodo de aprendizaje lo considero muy positivo y valioso para mí.

6.3. Conclusion uso API

Durante el desarrollo de este proyecto se ha intentado evaluar el API de José Alapont la cual ayuda a gestionar máquinas de estados, en esta obra esas máquinas han sido empleadas para definir el comportamiento de los enemigos, es decir construir una IA.

A fin de obtener una comparativa más fiable, se han desarrollado una de esas máquinas sin la API y otra con ella y se han tomado una serie de métricas las cuales nos ayudaran a dar una valoración objetiva.

Codigo SIN API

```
switch (m_tCurrentState) {
    case Tags.STATE_LOOKING_ENEMY:
        if( m_eEvent == Tags.EVENT_ENEMY_FOND ) {
            m_tCurrentState = Tags.STATE_ATTACKING;
        }
        if( m_eEvent == Tags.EVENT_LIFE_BELOW_50 ) {
            m_tCurrentState = Tags.STATE_DEFENDING;
        }
        if ( m_eEvent == Tags.EVENT_DEAD ) {
            m_tCurrentState = Tags.STATE_DEAD;
        }
        break;
    case Tags.STATE_ATTACKING:
        if( m_eEvent == Tags.EVENT_ENEMY_NOT_FOUND ) {
            m_tCurrentState = Tags.STATE_LOOKING_ENEMY;
        }
        if( m_eEvent == Tags.EVENT_LIFE_BELOW_50 ) {
            m_tCurrentState = Tags.STATE_DEFENDING;
        }
        if ( m_eEvent == Tags.EVENT_DEAD ) {
            m_tCurrentState = Tags.STATE_DEAD;
        }
        break;
    case Tags.STATE_DEFENDING:
        if( m_eEvent == Tags.EVENT_LIFE_BELOW_25 ) {
            m_tCurrentState = Tags.STATE_DEFENDING;
        }
        if ( m_eEvent == Tags.EVENT_DEAD ) {
            m_tCurrentState = Tags.STATE_DEAD;
        }
        break;
    case Tags.ACTION_RUNNING:
        if ( m_eEvent == Tags.EVENT_DEAD ) {
            m_tCurrentState = Tags.STATE_DEAD;
        }
        break;
    case Tags.ACTION_DEAD:
        break;
    default:
        break;
```

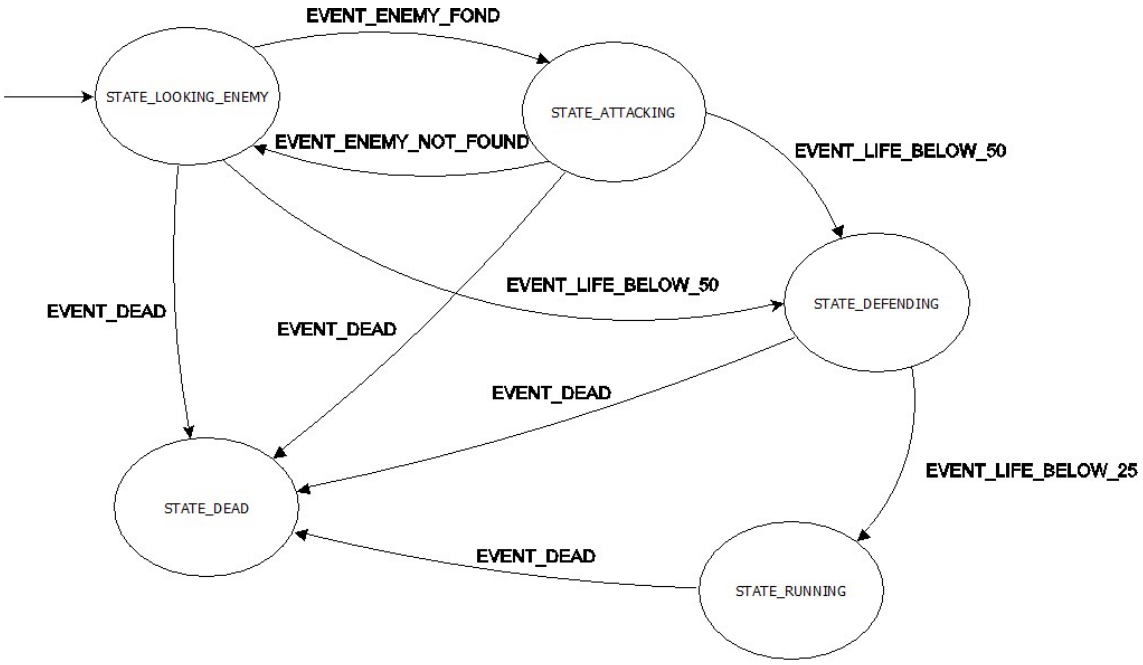
Codigo CON API

```
<Fsm>
    <Callback>CheckEvents</Callback> <!--Method for eve
    <States>
        <State Initial="YES">
            <S_Name>STATE_LOOKING_ENEMY</S_Name>
            <S_Action>ACTION_LOOKING_ENEMY</S_Action>
            <S_inAction>NULL</S_inAction>
            <S_outAction>NULL</S_outAction>
            <S_Fsm></S_Fsm>
        </State>
        <State Initial="NO">
            <S_Name>STATE_ATTACKING</S_Name>
            <S_Action>ACTION_ATTACKING</S_Action>
            <S_inAction>NULL</S_inAction>
            <S_outAction>NULL</S_outAction>
            <S_Fsm></S_Fsm>
        </State>
        <State Initial="NO">
            <S_Name>STATE_DEFENDING</S_Name>
            <S_Action>ACTION_DEFENDING</S_Action>
            <S_inAction>NULL</S_inAction>
            <S_outAction>NULL</S_outAction>
            <S_Fsm></S_Fsm>
        </State>
        <State Initial="NO">
            <S_Name>STATE_RUNNING</S_Name>
            <S_Action>ACTION_RUNNING</S_Action>
            <S_inAction>NULL</S_inAction>
            <S_outAction>NULL</S_outAction>
            <S_Fsm></S_Fsm>
        </State>
        <State Initial="NO">
            <S_Name>STATE_DEAD</S_Name>
            <S_Action>ACTION_DEAD</S_Action>
            <S_inAction>NULL</S_inAction>
            <S_outAction>NULL</S_outAction>
            <S_Fsm></S_Fsm>
        </State>
    </States>
```

Para dicha evaluación se van a aplicar las siguientes métricas.

- Numero de lianas de código utilizadas parea realizar la implementación.
- Tiempo de implementación aproximado para el primer FSM.
- Tiempo de implementación aproximado para Nº FSMs de una complejidad similar.
- Mantenibilidad de los FSM
- Escalabilidad de los FSM.

El FSM escogido para esta comparativa un FSM determinista que controla el movimiento de un enemigo, podemos verlo en el grafo mostrado a continuación.



El comportamiento de este grafo esta explicado en la obra, en este caso nos centraremos más en la comparativa de implementaciones con y sin API de gestión de máquinas de estado.

6.3.1. Número de líneas

En esta métrica se mide el número de líneas totales que se han utilizado para el desarrollo de la IA, para este conteo no se han tenido en cuenta los archivos XML utilizados para la API ni ningún archivo externo a la clase IAMechBasic.cs ya que en esta clase se concentra la lógica implementada para esta funcionalidad.

Número de líneas utilizadas **sin el API**: 190 líneas

Número de líneas utilizadas **con el API**: 140 líneas

6.3.2. Tiempo de la creación del primero FSM

En esta métrica se mide el tiempo en el que se tardó en desarrollar y testear la primera implantación con las dos técnicas.

Tiempo utilizado para implementar la implementación **sin el API**: 6h

Tiempo utilizado para implementar la implementación **con el API**: 9h

6.3.3. Tiempo de la creación de Nº FSMs

En esta métrica se mide el tiempo en el que se tardó en desarrollar y testear la implantación con las dos técnicas, estos tiempos son aproximados ya que no se ha implementado tantas veces como para que este valor sea fiable al 100% aun así es un dato muy interesante.

Tiempo utilizado para implementar la implementación **sin el API**: 3h

Tiempo utilizado para implementar la implementación **con el API**: 2h

6.3.4. Mantenibilidad de los FSM

En esta métrica se mide como es de mantenible del código obtenido tras la implementación del FSM. Se ha valorado con un número del uno al cinco, siendo uno el más bajo y cinco el más alto la capacidad de ser mantenible. Es decir, cinco será el mejor resultado posible. Se han tenido en cuenta para la evaluación de esta métrica la capacidad de resolver problemas surgidos en el desarrollo de los FSM.

Valor obtenido para la implementación **sin el API**: 3

Valor obtenido para la implementación **con el API**: 4

6.3.5. Escalabilidad de los FSM

En esta métrica se mide la capacidad de escalabilidad que disponen los FSM. Se ha valorado con un número del uno al cinco, siendo uno el más bajo y cinco el más alto la capacidad de ser escalable, es decir cinco será el mejor resultado posible. Se ha tenido en cuenta para la evaluación de esta métrica como quedaría el código con FSM más complejo o más grande.

Valor obtenido para la implementación **sin el API**: 2

Valor obtenido para la implementación **con el API**: 4

6.3.6. Valoración general

Tras valorar los datos obtenidos tras las métricas se puede afirmar que la API para manejar máquinas de estado desarrollada por José Alapont proporciona ciertos beneficios respecto a la no utilización de esta.

Aun así, cabe destacar ciertos puntos, la API no solo nos proporciona un conjunto de herramientas que sabemos que funcionan inherentemente, sino que otorga un método de trabajo que obliga a estructurar de cierta manera el código.

En este caso esa estructura creo que me ha aportado valor ya que me ha ayudado a plantear de una manera más acertada la problemática. Es verdad he tenido un pequeño coste en horas, pero ha merecido la pena.

Por otra parte, el FSM escogido para la comparativa es bastante fácil de implementar sin la API. Tal vez si se hubiera realizado la comparación con el FSM probabilístico que se usa para el comportamiento del otro Mech los resultados le hubieran favorecido todavía más.

El uso de la API ha sido un completo acierto, ya que ha facilitado en gran medida el desarrollo de la obra.

Aunque no se ha llegado a alcanzar el máximo potencial al que se le podría sacar a la API ya que tiene un gran abanico, con las que se han llegado a utilizar han sido más que suficiente para el proyecto.

Si se tuviera que sacar alguna pega al sistema propuesto por José Alapont es que el sistema de eventos que esta implementado, tan solo es capaz de capturar un solo evento por interacción y me hubiera gustado que por sí solo tuviera una cola interna de sucesos de este tipo, de esta manera se podrían lanzar dos eventos consecutivos y el FSM sería capaz de cambiar dos veces de estado en la misma interacción.

6.4. Problemas surgidos

Durante el desarrollo se han ido encontrando algunos problemas que se han intentado mitigar o resolver. Hacer una pequeña reflexión sobre estos puntos puede ayudar a uno mismo e incluso a otros compañeros, a tomar medidas preventivas para evitar dichos problemas.

Por eso a continuación se van a destacar algunos de los problemas que han ido surgiendo en el proyecto:

6.5. Gestor de versiones

Una de las herramientas que se utiliza en el proyecto ha sido un gestor de versiones, concretamente Mercurial. Esta herramienta es fundamental a la hora del desarrollo del software. En estos tiempos permite llevar un control de las versiones que se han ido haciendo de los ficheros. Cuando más potencial se le puede sacar a estas herramientas es cuándos e trabaja en un equipo de desarrollo ya que es capaz de fusionar (hacer *merge*) de código siendo capaz de gestionar los cambios realizados por dos o más personas del equipo sin pérdida de información por ninguna de las partes, esto se consigue subiendo toda esa información a un servidor conjunto, todo esto se consigue trabajando con estas herramientas.

Puede sonar un poco raro cuando se habla de problemas de un gestor de versiones cuando solo ha trabajado una persona en el proyecto. Se ha intentado aprovechar el tiempo al máximo en todos los lugares donde he estado y muchas veces he trabajado con equipos diferentes sobre todo con mi portátil y mi sobre mesa, con que lo podríamos decir que he trabajado como si estuviese colaborando con otra persona, aunque en este caso era uno mismo.

Cuando se empezaba una tarea con un dispositivo no se subía al gestor de versiones hasta que esta estuviera acabada, de esta manera es como suelen trabajar los equipos en estos proyectos. Está totalmente prohibido en cualquier proyecto software subir código al servidor que no esté funcionado correctamente de manera consciente, es decir dejando las cosas a mitad, por ejemplo.

Especialmente *Unity* es bastante exigente respecto a colaboración de código porque esta herramienta genera múltiples archivos temporales además de que las escenas se guardan por

defecto en un formato binario con lo que la comparación de cambios para el gestor es muy complicada por lo que cuando fusiona el código del lado del servidor con el que tienes en local habitualmente genera muchos errores y por último la configuración de los *layouts* que es lo que nos permite tener el motor con las ventanas que nos gustan y de los tamaños que queremos, esta configuración viene muy marcada por la resolución de pantalla con la que nos encontremos en un momento determinado y toda esa información también se almacena en el proyecto con lo que se sube al servidor.

Todas las razones mencionadas anteriormente son las que dificultan el uso del gestor de versiones para un video juego creado con *Unity,* a continuación, voy a mencionar las soluciones que se han podido encontrar para solventar estos problemas.

Gitignore, por norma general, en la mayoría de proyectos software no se desea subir todos los archivos que se encuentran al gestor de versiones. Los archivos de configuración de editores, librerías que se descargan automáticamente, archivos del sistema, etc. son ejemplos de archivos que no deberían de estar en almacenados en el repositorio. Los gestores de versiones ya están preparados para este tipo de problemas. Se trata del archivos "*.gitignore*" este archivos permite evitar que ciertos archivos sean compartidos por el gestor de versiones. En los foros de oficiales de *Unity* podemos encontrar varios hilos que hablan sobre este problema y aportar soluciones bastante completas.

Ficheros "serializables", como bien se ha mencionado anteriormente algunos ficheros del motor se guardan en formato binario de manera predeterminada con lo que provoca fallos ahora de combinar código de manera automática.

En la ventana de *settings* del motor podemos encontrar las opciones que se ven en la imagen, una de estas opciones es "*Asset Serializable",* que tiene el valor "*Mixed*" esta opción se tiene que cambiar por el valor "*text scene format*" a partir de este momento los *assert* se guardaran en formato texto y con esto se consigue que el gestor de versiones pueda hacer la combinación de este tipo de ficheros de una manera más eficiente sin generar tantos errores.

Estos dos consejos que se acaban de mencionar son capaces de resolver este tipo de problemas el 90% de las veces. Si estos puntos no son resueltos al principio del proyecto, si haces una retrospectiva te das cuenta de que tienes muchas horas perdidas en el desarrollo. Sin mucho esfuerzo puedes encontrar en internet mucha gente que hable sobre estos problemas. Sin lugar a

duda este suele ser un error de principiante de esta tecnología que todo el mundo ha pasado por él y que solo puedes solucionarlo si ya te has visto en esa misma situación.

6.6. Versión 5 de *Unity*

Todo el proyecto ha sido desarrollado en Unity 4. Durante ese proceso, la compañía que provee este producto, lanzó una versión estable de la versión Unity 5 la cual aporta muchas mejoras, siempre es bueno trabajar con las últimas versiones ya que de esta manera se mantiene siempre con las mejores prestaciones del producto.

Cuando se intenta cargar un proyecto de la versión 4 en el motor de la versión 5, por la cantidad cambios que existen entre estas entre estas versiones, el propio motor, en el mejor de los casos es capaz de convertirlo sin problemas, en el peor de los casos toca rehacer parte de código o se pierde cierta información de algún *asset*.

Se hizo este proceso y en principio todo funcionó bien y quedé maravillado con lo fácil que había sido, por lo que se siguió trabajando con esa versión durante un tiempo. Al cabo de un tiempo llegó el momento de añadir la IA de los Mech al proyecto y para ello se hace uso de la API de José Alapont. Esta API no tiene compatibilidad con la versión 5 de *Unity*.

Al haber hecho ya la importación del proyecto a la versión 5 automáticamente se pierde la retro compatibilidad con la versión 4, dicho de otro modo, no podía volver a la versión 4.

Llegados a este punto tan solo se tenían dos opciones, actualizar el API de José Alapont para poder usarla en un proyecto con la versión 5 de *Unity* o recuperar toda la información del gestor de versionado antes de hacer el cambio de versión y aplicarle nuevamente los cambios que se habían realizados hasta la fecha.

Al final se decidió recuperar la versión 4 de *Unity* y volver a aplicar todos los cambios, ya que la mayoría de los cambios que habían realizado a partir de la migración de versión estaban concentrados en los *scrips*. Esto provocó que se perdiese información relevante de los *asset*, contando que ya se sabían los cambios que eran porque previamente se habían realizado, se tardaría menos que investigar acerca de los cambios que se tendrían que hacer a los *scrips* que se encuentran en la API.

Sabiendo todo esto no creo que realice ningún cambio de versión a mitad de un proyecto de estas características a no ser que sea obligado por alguna característica de esta versión que es vital para el proyecto o algún fallo que impida continuar con el desarrollo.

7. Trabajos futuros

El video juego que se llevado a cabo ha sido un alpha. Existen muchos puntos en los que este juego sería ampliable para futuros trabajos. En este punto se mencionarán los que se consideran más importantes y relevantes de cara a que pudiesen completarlo otras personas.

- Se podrían implementar nuevos gráficos y sonidos con los que mejorar la experiencia de usuario, a pesar de que técnicamente está bastante completo gráficamente aún queda mucho trabajo.

- Actualmente no se puede guardar el proceso de la partida, se podría implementar algún sistema de guardado tanto en la nube como en local.

- El modo multijugador, este proyecto está bastante encarado para la implementación de esta opción, de echo en el propio trabajo es destacado en varias ocasiones.

- Mejora de los Mech, actualmente no existe ningún mecanismo de mejora de Mech. Previamente se debería de integrar alguna manera de conseguir recompensas tras las victorias contra otros jugadores y gracias a esas recompensas poder mejorar el Mech.

Estas son las opciones que podrían ser interesantes, pero ahora mismo están implementadas las funcionalidades básicas del juego. Se podría dar una vuelta de tuerca a todo el concepto y generar otro juego a partir de este reutilizando las mecánicas ya implementadas.

Haciendo referencia al API de José Alapont, es una herramienta muy útil, pero se le puede sacar mucho más potencial del que tiene. Es una gran base de un proyecto muy interesante que espero que no se quede abandonado por el paso del tiempo.

- Una de las mejoras que es bastante clara sería crear un entorno gráfico que pudiera generar los XML de manera rápida y sencilla, esto haría que la escalabilidad de FSM más complejos mejorase en gran medida además con una herramienta visual de este tipo sería más fácil la detección de errores.

- Cuando se tiene una clase a la que quieres implementar el FSM se tienen que crear muchas variables y métodos que resulta un poco tedioso, si toda esa lógica referente a las máquinas de estado estuviera almacenada en una clase, su utilización sería más práctica.

- Para implementar esta API requiere tener un método *StringToTag* que resulta súper tedioso mantenerlo y no aporta nada a quien está haciendo uso de dicha API suprimir el requisito de implementar este método mejoraría la API.

- Estaría bien poder pasar una lista de eventos sucedidos y que sea capaz de tratarlo como una serie de eventos sucedidos secuencialmente para efectuar los cambios de estados. Esto facilitaría su uso y le daría más flexibilidad cara a otros proyectos. Ya que se puede dar el caso de que se generen dos eventos en un intervalo de tiempo muy reducido.

- La última versión de Unity es la 5, este API no funciona con dicha versión, esa versión de Unity incorpora mejoras significativas para el motor y sería interesante se puede usar en esta nueva actualización del motor.

Considero que estas, mejoran la escalabilidad el mantenimiento y la productividad a la hora de utilizar esta API mejorarían considerablemente. Y no solo eso, al crear una interfaz gráfica podría dar lugar a que no solo programadores hagan uso de ella sino personas con menos conocimientos técnicos pudieran participar en el desarrollo de FSM.

8. Bibliografía

José Alapont Luján (septiembre de 2014)

Título del proyecto: API de gestión de Inteligencia Artificial basada en las Máquinas de Estados Finitos en C#

Referencia:
https://riunet.upv.es/bitstream/handle/10251/47429/TFM%20Jos%C3%A9_Alapont_Luj%C3%A1n.pdf?sequence=1

9. Agradecimientos

Quisiera poner de manifiesto mis más sinceros agradecimientos a algunas personas que gracias a ellas de una forma u otra han hecho posible la presentación de esta obra.

En primer lugar, al Dr. D. Ramón Mollá Vayá, director de este proyecto por su apoyo constante y por empujarme con todas sus fuerzas en el último tramo de este camino.

También a José Alapont por haber realizado un trabajo de master magnífico capaz de desembocado en este otro.

Y por último a mi familia y amigos por darme ánimos y todas sus fuerzas, en especial a Mª Ángeles, Alfonso y Gema.

Mis más sinceros agradecimientos, sin todos vosotros esto no sería posible.

2016

TFG Anexo GDD

Autor:
Abel, Valero Jiménez

Director:
Dr. D. Ramón Pascual, Mollá Vayá

[GDD ANEXO]

Versión 1 | 01/09/2016

Plataforma

El proyecto está enfocado a usarse en dispositivos móviles, concretamente para la plataforma Android.

Dado que la realización de este proyecto será enteramente en el programa *Unity*, será fácil exportarlo a diversas plataformas.

Título

Mech Battle

Género

Es un juego de estrategia por turnos en el que un turno se compone por varias acciones que puede hacer el jugador en cada uno de estos (disparar, moverse, rotar).

Temática

Un mundo mucho más avanzado que el nuestro, donde los combates que realizan las máquinas son los reyes del entretenimiento, todo este mundo gira entorno a estos combates. Todo el mundo sueña con tener su propio robot invencible capaz de derrotar a cualquier oponente, y con ello ganar reputación para así algún día poder participar en la WMT (World Mech Tournament)

Son robots controlados por personas, que dan la posibilidad de demostrar el ingenio y la habilidad del jugador sobre el campo de batalla.

Target

Hombres de 15 – 30, los juegos de estrategia están dominados por el público masculino. La edad de disponer un smartphone es cada vez más baja y se puede tener la edad del target más baja. Es un juego para un público joven pero tampoco para niños ya que requiere que el usuario plantee estrategias elaboradas.

Descripción general

El juego consiste en un enfrentamiento de todos contra todos, donde todos los jugadores luchan entre sí para dominar la arena, y cada jugador podrá elegir unas acciones determinadas: atacar, moverse, esperar, rotar... Es un juego por turnos, donde todos los jugadores tienen un tiempo para elegir un número determinado de acciones, 5 exactamente por turno; una vez todos los jugadores tengan sus acciones elegidas se ejecutarán todas en el orden que ha establecido cada jugador de forma simultánea, en el mismo intervalo de tiempo para todos.

Es un juego donde premia la estrategia, y el azar a partes iguales. Por una parte tendrán que predecir los movimientos de sus rivales cosa que no será nada fácil y por otra parte se juega con

un mapa dinámico que da ese toque de aleatoriedad que tanto gusta a los amantes de este género.

Los jugadores pueden intentar que los enemigos se ataquen entre si para tan solo tener que rematarlos en los momentos en los que estén mas débiles o destrozarlos con grandes ataques, porque lo importante es sobrevivir.

StoryLine

En un futuro lejano, un jóven ingeniero ha encontrado un robot tirado en la basura y ha decidido repararlo para así obtener su primer robot de combate. Todos los días sueña con ganar el "World Mech Tournament" o también llamado WMT, el más prestigioso de los torneos de robots del mundo en la que otros brillantes ingenieros construyen sus propios robots para demostrar quién es el mejor estratega de la toda la liga.

Las batallas de robots tienen lugar en el enorme estadio de Arcam con un avanzado escenario, en el que se han recreado mapas y laberintos asombrosos, desde bosques flotantes hasta alcantarillas post-apocalípticas llenas de sorpresas y mejoras temporales. Para ello los robots deberán abrirse paso para conseguirlas, sortear y lograr sobrevivir y así poder aplastar al el resto de los otros robots para conseguir la victoria.

Nuestro joven ingeniero deberá aprender a dominar su nuevo robot le esperan muchas horas de duro entrenamiento tendrá que conocer los mapas para así poder sacar ventaja de ellos y esquivar las terribles trampas que estos esconden y sobretodo conocer a sus rivales, nunca sabes que pequeños trucos pueden tener esas terroríficas máquinas.

Si es capaz de conseguir todo esto, podrá llegar a ser imparable y conseguir ser el dueño y señor de la arena para poder hacer su sueño realidad: ser el gran campeón del World Mech Tournament y formar parte de las leyendas de su época.

Estética

Dado que la naturaleza del juego es, en cierto modo, arcade, hemos decidido utilizar una estética que conmemora a este género; por ello, utilizamos el lenguaje que utilizaban en los inicios de los videojuegos: El pixel art. Se utiliza una paleta viva, pero no saturada, y para destacar ciertos elementos, se utilizan colores neón con un vista cenital con el frente ligeramente forzado.

Esto no será posible para la primera versión del juego se utilizara un surdido reducido de imágenes y sonidos cogidas de internet adaptadas a este proyecto, ya que no se dispone de diseñadores que puedan hacer un desarrollo a medida.

Las imágenes y sonidos que se han utilizado aparecen en los créditos del juego el lugar donde se han obtenido.

Referencias

Hoplite - https://play.google.com/store/apps/details?id=com.magmafortress.hoplite&hl=es

Gran juego de estrategia por turnos en el que turno a turno se tiene que competir contra enemigos hasta limpiar por completo la sala, en él el objetivo es llegar a las salas más profundas donde cada vez te atacan más enemigos. Es un juego en el que el personaje va mejorando y ganando habilidades poco a poco. Se han tenido muy presente sus grandes virtudes a la hora de pensar muchas de nuestras ideas.

Flamberge - http://www.indiedb.com/games/flamberge

Sin lugar a duda, este juego plasma totalmente la idea principal a la que se queríaqueríamos llegar. Nuestro juego sería una versión más restrictiva respecto al movimiento y las acciones. Creemos que esto simplifica el juego y lo hace más fácil para jugadores inexpertos aunque sin perder la diversión a los jugadores más veteranos.

Acero puro - https://es.wikipedia.org/wiki/Real_Steel

Es una película del año 2011 de ciencia ficción. La película se argumenta en mundo donde los seres humanos han sustituido por robots el boxeo, todas estas peleas provocan un gran reclamo para la sociedad. Nos gustaría crear una esencia similar a la de esta película donde todos sueñan en tener su propio robot de lucha y por su puesto competir en la WRB (World Robot Boxin)

Pokemon - https://www.pokemon.com/es/

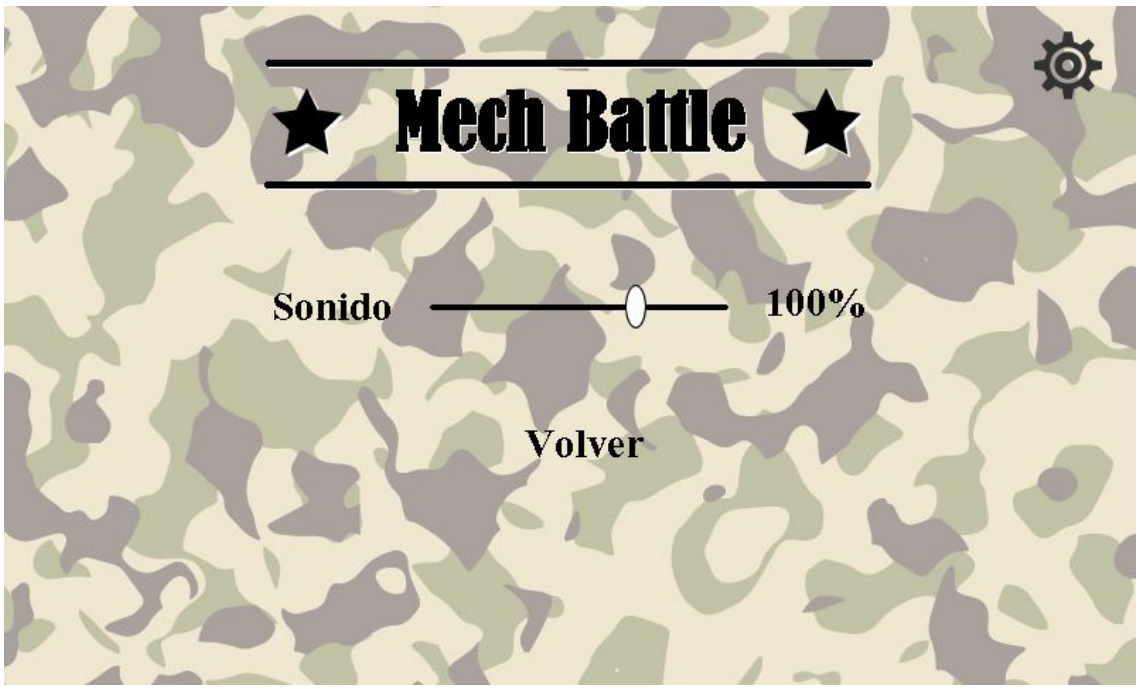
Sin duda un juego que marcó tendencia y que revolucionó los juegos de hasta ese momento. Es un RPG en el que el personaje principal está deseoso de cumplir su sueño que es ser entrenador pokemon para ganar la liga pokemon. Es un gran referente para jugadores de los 90 y tanto la ambientación como algunas mecánicas como la liga o la evolución del robot se han visto influenciadas por este maravilloso juego.

Interfaz del usuario

Menú principal

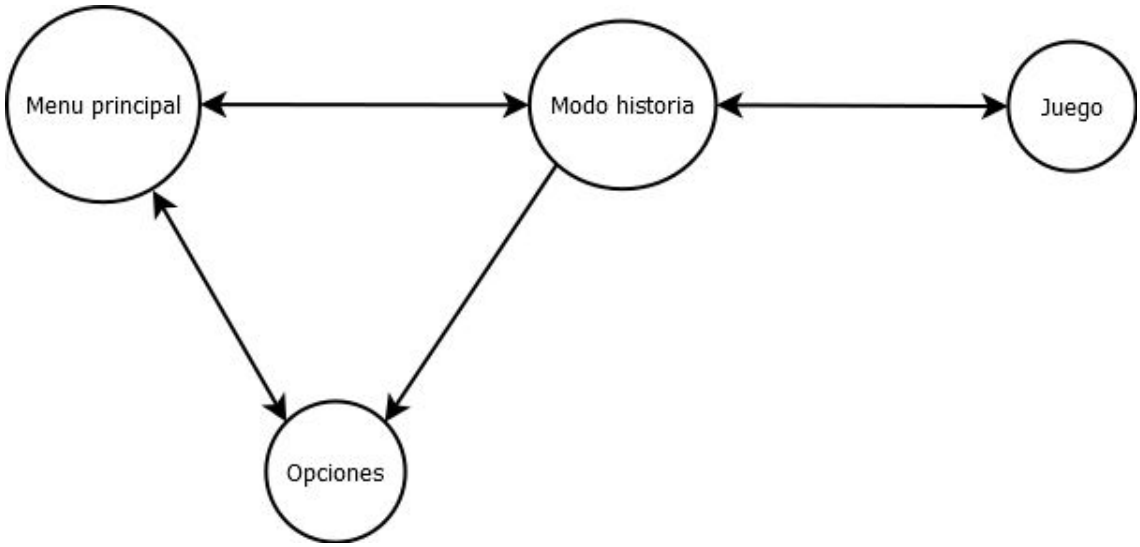
El menú principal es un menú tradicional. El control del menú es táctil y consta de un listado con los distintos modos jugables. Consta de: el modo historia que será el modo de un jugador, modo online para jugar contra otros jugadores, el taller para poder mejorar mech para los modos de un jugador y multijugador, los créditos y salir. En la parte superior derecha se sitúa el menú de opciones. Las opciones tachadas del menú no estará disponible en el TFG.

Desde cualquiera de las pantallas del menú se puede acceder al menú de opciones en la parte superior derecha. Los ajustes son: Sonido, para regular el sonido de todo el juego.



Al iniciar el modo historia aparecerá el mejor tiempo, la mejor puntuación y el resumen del escenario, contando la historia y el trasfondo de la batalla que se va a realizar.

Se puede ver en el siguiente grafo el flujo de las pantalals del menu. Si iniciamos el juego por primera vez el punto de partida de este grafo sera el Menu Principal, si acabamos de jugar una partida el puto de partida sera el Modo Historia.



Jugabilidad

En todos los aspectos de la jugabilidad se ha tenido en cuenta que es un diseño para móviles. Las zonas de interacción para el usuario están destacadas en la Interfaz del usuario y son los suficientemente grandes para pulsarlas sin problemas. Es un juego intuitivo y fácil de jugar.

Tiene una experiencia de juego tranquila ya que se proporciona suficiente tiempo para poder designar tus acciones en cada ronda, además de permitirte modificarlas en cualquier momento antes de que estas se confirmen.

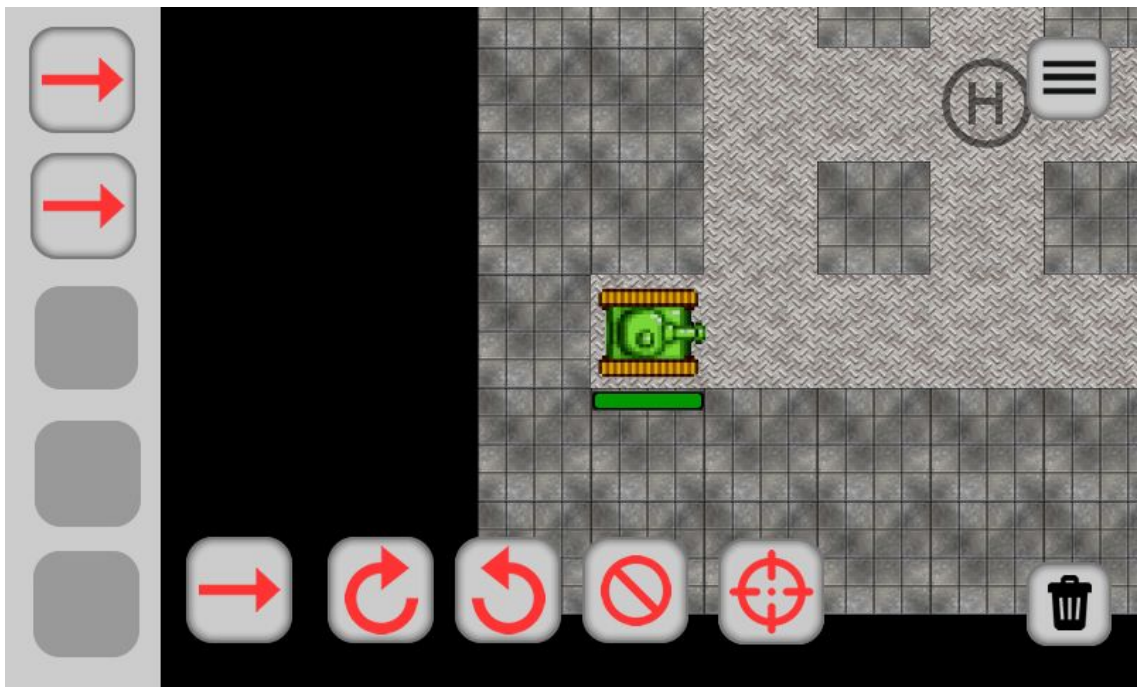
Interfaz de juego

Toda la interfaz está diseñada para ser usada en dispositivos móviles. Los botones son grandes y los elementos están colocados teniendo en mente que se jugará sobre dispositivos móviles los cuales carecen de pantallas grandes. Es decir, los elementos están distribuidos de manera proporcional según el tamaño de la pantalla, para que la jugabilidad no se vea afectada por la resolución del dispositivo.

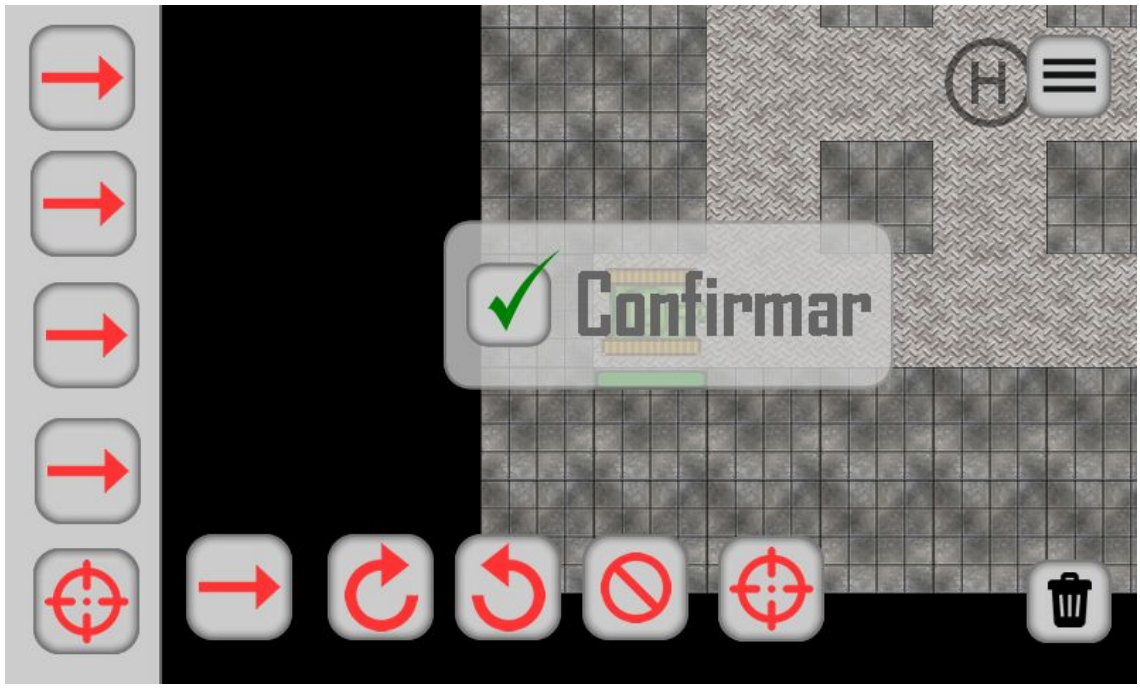
El uso de la interfaz es intuitivo, los elementos con los que puedes interactuar están destacados y se utilizan iconos que representan las acciones del personaje para facilitar el reconocimiento de las acciones.

La zona izquierda de la pantalla se reserva para las acciones ya introducidas por el usuario. El usuario podrá ver las acciones que ha introducido para ver cómo quedarán antes de que se confirmen de manera definitiva. El orden correcto para leer estas acciones es la parte superior. La primera acción que se llevará a cabo hasta la parte inferior de la barra de acciones, que será la última acción que realizará el personaje.

La parte inferior está dedicada a la selección de acciones a realizar, el usuario utilizará esta parte de la interfaz para introducir los comandos que llevará a cabo el Mech.



Una vez insertadas todas las acciones aparecerá un mensaje en pantalla para poder confirmar las acciones introducidas, si se desea confirmar se le pulsará al nuevo botón situado en la parte central y si se desea cancelar, se podrá borrar utilizando el icono de la papelera.



En la parte superior derecha estará el botón dedicado al menú, que dará la opción de continuar con la partida y abandonar juego para regresar al menú principal.

Tras finalizar la partida aparecerá un panel con el que se podrá ir al menu principal y se podrá ver el resultado de la partida.

Hub

Modo historia - En este modo de juego dispondrá de 5 niveles donde te enfrentaras contra la IA y se podrá disfrutar de la historia de nuestro Mech y hasta dónde se puede llegar con él.

Modo Online - Te enfrentarás a otros jugadores este modo requiere conexión a internet. (Objetivo no TFG)

Taller - En este taller es donde se cambiarán las piezas de repuesto por mejoras de personaje y cambios de aspectos. (Objetivo no TFG)

Opciones - En este apartado podrás ajustar: El Volumen música, el Volumen efectos, Resetear el juego.

Créditos - En este apartado podrás ver las personas que han colaborado en este proyecto.

Controles

Se utiliza el control táctil de la pantalla del dispositivo móvil. El jugador podrá elegir las acciones a tomar por el robot usando los iconos disponibles en la zona inferior de la pantalla del juego, estas se podrán cancelar en cualquier momento utilizando los botones de cancelación en la parte superior de la pantalla. La cámara irá anclada al mech del jugador para dar dificultad, a medida que se mueve el mech del jugador, la cámara lo seguirá.

Modo historia

El modo historia es donde los jugadores competirán contra la IA. Antes de empezar cada partida aparecerá un texto que es lo que ha motivado al protagonista del juego ha realizar su próximo combate en Arcam.

El modo historia se compone de 10 combates, pero para el alcance el proyecto tan solo estará disponible el primero.

Primer combate

Texto: Aquí empieza tu camino joven, tendrás que luchar contra 3 novatos como tú, tal vez con un poco de suerte tu pequeña chapucilla es capaz de sobrevivir un par de minutos. No te deprimas si mueres nada más empezar, no todo el mundo ha nacido para ser un Ingeniero, el mundo también necesita granjeros. Te voy a dar un consejo, no te fíes de nadie; ya estás en la arena, aquí no tienes ningún amigo, cuando menos te lo esperes alguien te clavará un destornillador en la espalda.

Dificultad: Fácil.

Enemigos: Cantidad 3, Salud 100p, Fuerza 9.

Recompensa: Nada.

Moneda

(Objetivo no TFG) La moneda de este juego serán las piezas de repuesto con estas piezas podrás evolucionar tu Mech y comprar nuevos modelos en el taller, podrás conseguir estas monedas de varias maneras:

- Compartiendo el juego por tus redes sociales +250 .
- Jugando en cualquiera de los modos.
    - Ganar un enfrentamiento + 100
    - Eliminar a otro jugador + 50
- Comprándolo con dinero real a través de la pasarela de pago de google play.
    - 1000 = 1.95€
    - 3000 = 4.95€
    - 6500 = 9.95€

Puntuación

Tras finalizar una batalla se podrá ver la puntuación total obtenida, de esta manera se podtrá tener constancia de tus grandes esfuerzos durante la lucha y poder compararla con otras puntuaciones, ya sean tuyas o de tus amigos. Sistema de puntuación:

- Ganas 10 puntos por sobrevivir cada ronda.
- Ganas 20 puntos cada vez que dañas a un rival.
- Ganas 50 puntos por eliminar a un rival.
- Ganas 5 puntos por coger un Power UP.
- Pierdes 10 puntos cada vez que eres dañado.

Personajes

Mech - Son las máquinas que participan en los combates, estas pueden estar controladas por los jugadores o por la IA. Estas máquinas disponen de una cantidad de vida y una potencia de fuego que va aumentado según el nivel de mech.

Existen 2 tipos de Mech, el que se controla y el resto. Estos están diferenciados por los colores de los Mech.

Icono	**Decripción**
	Mech enemigo.
	Mech que se maneja.

Fases de un combate

1) Empieza la partida - Selecciona el modo de juego para poder empezar la batalla.

2) Seleccionar las acciones - Cada Mech dispone de 5 acciones que podrá realizar en su turno, estas acciones se pueden repetir y disponer en el orden que quiera cada jugador. En modo online habrá un tiempo limitado para realizar esta elección, si no se realiza a tiempo el Mech no se moverá en todo el turno.

3) Confirmar acciones - Una vez creas que tus acciones son las ideales para la situación confirma las acciones para que comience su ejecución.

4) Ejecutar acciones - Cada acción elegida por ti se ejecutará secuencialmente para tu Mech, simultáneamente se ejecutarán las de los otros Mechs.

5) Si todavía quedan oponentes volver a la fase 2.

6) Fin de la partida - Mostrará el resultado del combate, junto con la puntuación obtenida.

Elementos

Mapa - Es el escenario del juego, por lo tanto lugar por donde los Mechs podrán moverse, aunque habrá zonas del mapa que estarán cubiertas de obstáculos (piedras, árboles, barrancos… etc) y por ellas los Mechs no podrán moverse.

Todo el mapa está cubierto por una cuadrícula en la que el usuario podrá diferenciar claramente sus dimensiones. Las acciones de movimiento van determidas por esta cuadrícula.

Imagen	Descripción
	**Pared** - Este es un obstáculo. Ni mech, ni sus disparos podrán atravesarlo.
	**Suelo** - Esta chapa metálica representa el suelo y será por donde podrán moverse los mech.
	**Cinta de movimiento** - Están representadas en el mapa en una posición de la cuadrícula. Este elemento efectuará inmediatamente su acción en la dirección que apunta la flecha de la casilla independientemente de las acciones que previamente se establezcan al patrón de movimiento del Mech, si un mech está en la posición de la cinta de movimiento este será desplazado hacia donde indica la flecha. Ejemplo: Si un Mech se posiciona sobre una cinta de movimiento y su siguiente acción era disparar, el mech se moverá hacia la dirección de la cinta y disparará al mismo tiempo.
	**Zona de spawn** - Es la zona donde aparecerán los ítems potenciadores del juego. En ellos se pueden encontrar los escudos, reparaciones, etc. El tiempo en el que irán apareciendo estos ítems será aleatorio.
	**Barra de vida** - Es un indicador de la vida actual de un Mech.
	**Bala** - Son las balas que disparan los Mech.

Sprits

Imágenes	Uso
	Se usa en la cinta de movimiento
	Sea usa cuando tienen el item del escudo encima del Mech
	Se usa para los movimientos del Mech
	Se usa cuando una bala golpea a un Mech

Ítems

Se llamarán ítems a los objetos que hay distribuidos por el tablero de juego los cuales siempre aparecerán encima de una Zona de spawn. Estos ítems llamaran la atención de los jugadores por su brillo verde, se pueden coger y otorgarán beneficios a los jugadores que consigan cogerlos. Una vez cogidos se usarán automáticamente y desaparecerán del mapa.

Icono	**Descripción**
	Llave inglesa - Este ítem cura 40p de vida.
	Escudo - Este escudo te protege de un impacto.

Acciones

Llamamos acciones a las cosas que puede realizar un Mech. Estas acciones son elegidas por los jugadores y cada uno de los Mech podrá realizar 5 acciones en cada una de sus fases. Todas las acciones durarán el mismo tiempo, es decir, el Mech tardará lo mismo en realizar la acción de Moverse que la acción de Rotar hacia la derecha.

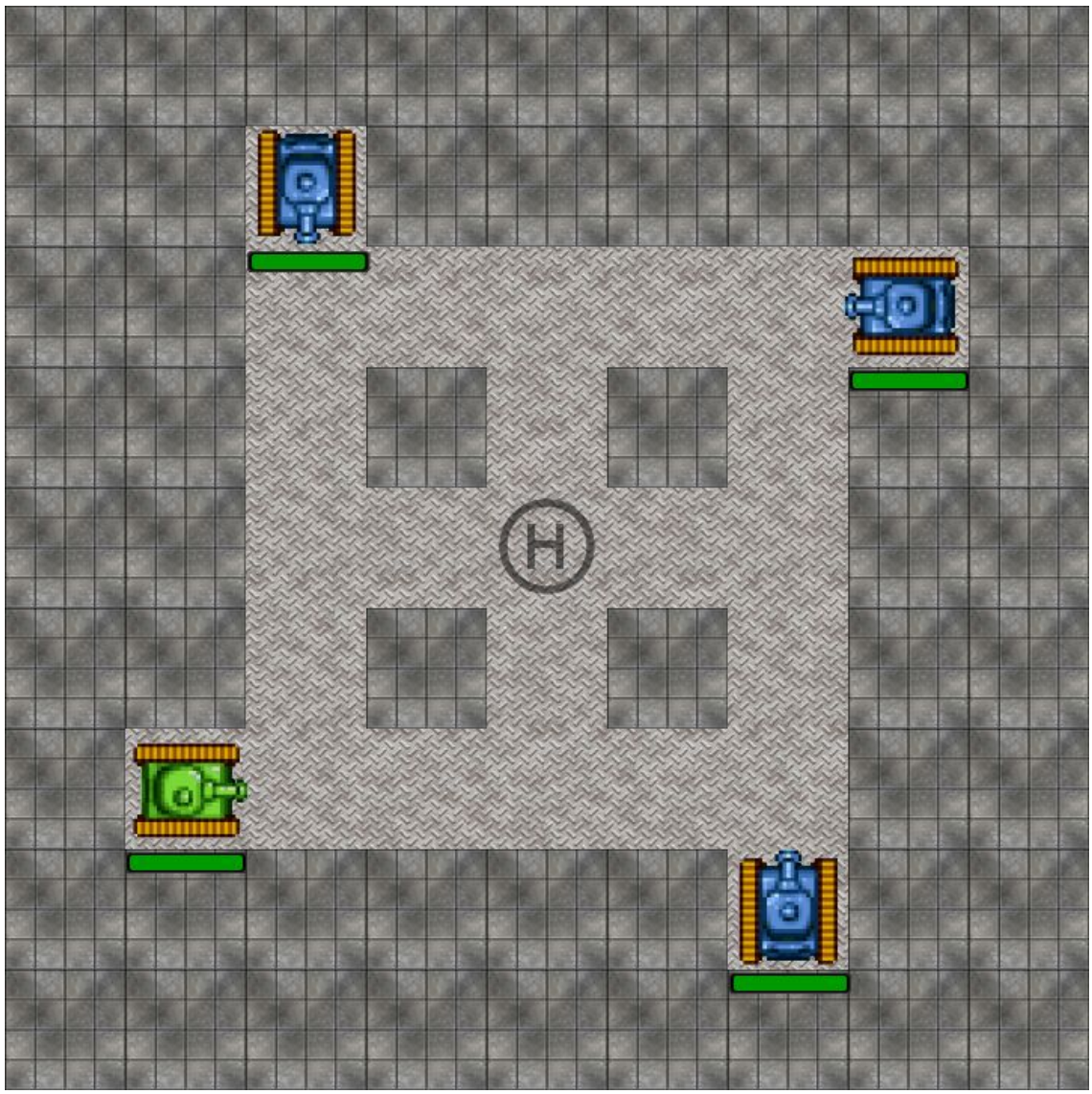
Iconos	**Descripción**
	Moverse - El Mech se desplazará una casilla hacia delante, es decir, en la dirección a la que apunta su cañón. Si dos o más Mech intentan realizar un movimiento a la misma posición en un momento determinado, el juego elegirá al azar uno de los dos como ganador y será el que ocupe esa posición, en cambio si un Mech intenta moverse a una posición en la que antes había otro Mech y este va a dejar libre esa posición, si se podrá realizar este movimiento y ocupar la casilla previamente ocupada.
	Rotar hacia la izquierda/derecha - Esta acción permitirá al Mech rotar 90º en la dirección que elija.
	Esperar - Cuando se elige esta opción el Mech no hará nada.
	Disparar - Cada Mech es capaz de disparar a una distancia determinada. (Esta acción se podrá potenciar en el Taller).

Mapas

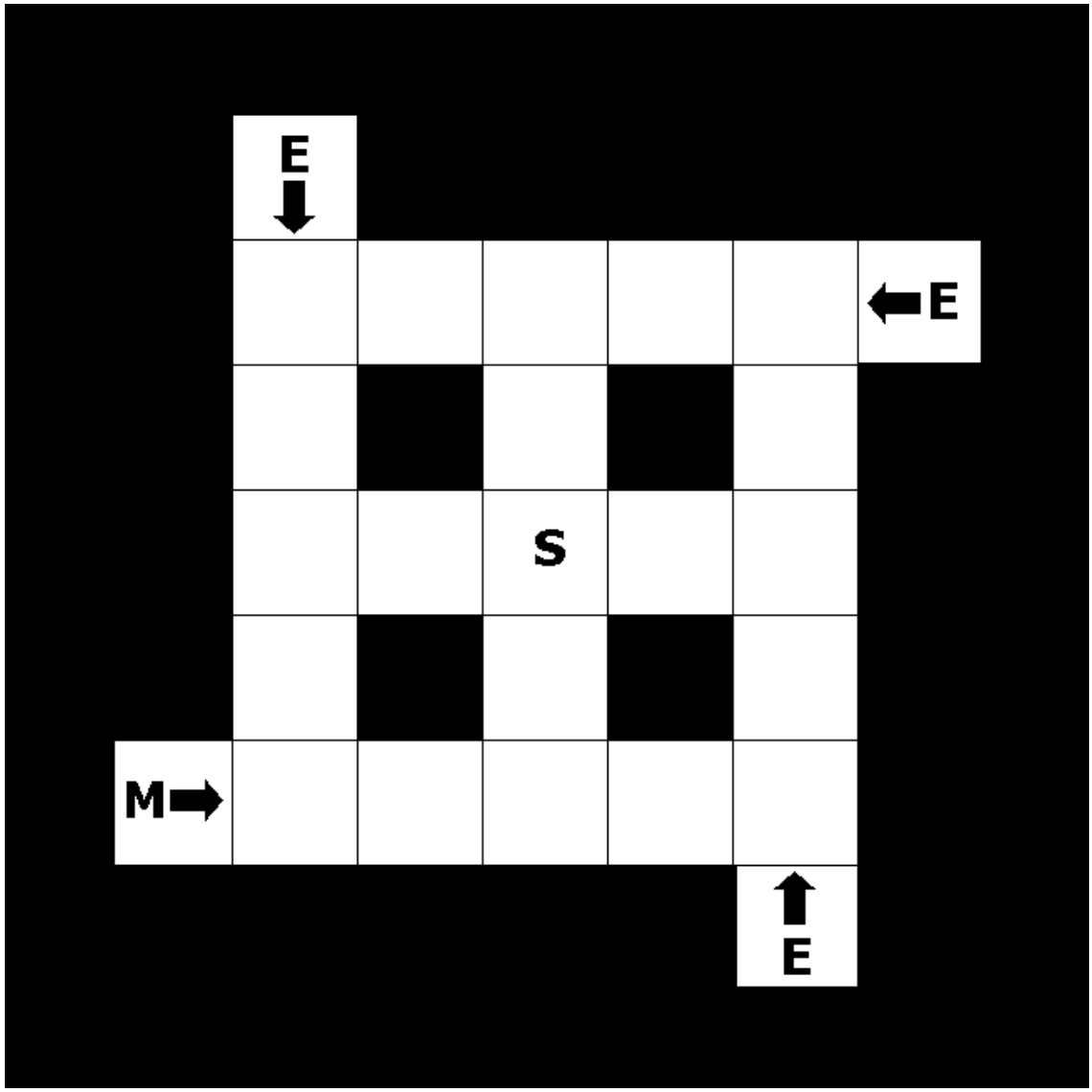
El terreno donde los Mech luchan entre ellos, lo llamamos arena, esta arena es muy importante ya que condicionará todas las acciones.

Este primer nivel fomenta la igualdad entre todos los Mech, es una arena pequeña pero muy entretenida porque los jugadores más avispados serán los que lleguen antes a conseguir el ítem que aparezca en la zona de spawn situada en la zona central del mapa. No será fácil ya que compiten contra otros tres jugadores que luchan por sobrevivir y apenas tienen espacio para correr. Este mapa fomenta la lucha entre los Mech y solo los más fuertes son lo saldrán vivos de esta arena.

Gráficos del mapa:



Esquema del mapa:



Leyenda:

M: El Mech que tu controlas.

E: Los Mech enemigos controlados por otras personas o la IA.

S: Zona de spawn.

→ : Cinta de movimiento.

Parte negra: Terreno no jugable.

Parte Blanca: Terreno jugable.

Música

Link	Uso
https://www.youtube.com/watch?v=ZYjg6QhWjLc&list=PL42255DF6EA4B55B5&index=3	En los combates.
https://www.youtube.com/watch?v=70M25NvQcpo&list=PL42255DF6EA4B55B5&index=11	En el menú.
https://www.youtube.com/watch?v=Ucsl0EGOVmM&index=10&list=PL42255DF6EA4B55B5	Cuando mueres en el combate.
https://www.youtube.com/watch?v=wzLmAF7MSuE	Cuando se mueve el mech.
https://www.youtube.com/watch?v=owF0SlHMRNE	Cuando el mech dispara.
https://www.youtube.com/watch?v=M27_cYTcZJI	Cuando un mech es alcanzado por un disparo.
https://www.youtube.com/watch?v=OrfHYxqH8UE	Cuando llevas puesto el power up de escudo.
https://www.youtube.com/watch?v=G0M2EM3BTOk	Cuando un mech con el escudo es alcanzado por un disparo.
https://www.youtube.com/watch?v=_E8aDr9DxSU	Cuando un mech coge el ítem Llave inglesa.
https://www.youtube.com/watch?v=_E8aDr9DxSU	Cuando sale un ítem en el spawn.