# Programación de Inteligencia Artificial avanzada en entornos *Unity*

Rubén Lozano Gómez

**Resumen**— El objetivo de este proyecto es el desarrollo de un videojuego en 3D, de estilo *shoot'em-up* con temática espacial, multijugador y compatible con periféricos (*gamepads)*. La idea principal es que el juego esté orientado a una jugabilidad profunda pero fácil de manejar y con vista cenital a pesar de que todos los modelos del juego estén en 3D. El enfoque principal del proyecto es el desarrollo de varios niveles de IA de diversa complejidad, así como implementar las mecánicas de juego necesarias para tener una experiencia completa de juego. Los algoritmos de IA utilizados se definirán a partir de un estudio profundo de los más usados en el sector de los videojuegos y más en concreto en los juegos de este género, buscando siempre la variable innovadora queriendo aportar una visión diferente a los videojuegos de este estilo. El resultado final del proyecto es un nivel jugable que consistirá en ir superar enemigos dotados de las diferentes IA desarrolladas a lo largo del proyecto de forma gradual.

**Palabras clave**— Videojuego, Inteligencia Artificial, Unity, Shoot'em-up, Programación de Videojuegos.

**Abstract**— The objective of this project is the development of a 3D videogame with a *shoot'em-up* space theme style, multiplayer, and with peripheral (*gamepads)* support. The main idea is a game oriented to a deep gameplay but easy to play with top view and 3D models. The project focus on the development of multiple AI levels with different complexity and the integration of game mechanics that are needed for a complete game experience. The AI algorithms will be defined by an intense study of the most used AI algorithms in the videogames sector and more specifically in the videogames of the same genre. The final result of the project is a playable level consisting on defeating enemies with the behaviour of the different AI that have been developed through the project gradually.

**Index Terms**— Videogame, Artificial Inteligence, Unity, Shoot'em-up, Game Programming.

## 1 INTRODUCCIÓN

### 1.1 MOTIVACIÓN

El objetivo de este proyecto es explorar las posibilidades de programación de diferentes niveles de inteligencia artificial dentro del contexto del desarrollo de Polarity, un videojuego para PC. Polarity pertenece al género *shoot'em up* [*1*] y es cooperativo para dos jugadores. La sincronización entre ambos jugadores es vital para superar los obstáculos que se proponen y sobrevivir en un mundo repleto de enemigos.

Polarity es un juego de un género considerado nicho dado que no son muchos los juegos de este estilo que salen de Japón [*2, 3, 4, 5*], donde hay más mercado. Precisamente por esto se trata de un juego que va a tener poca competencia y que puede tener hueco en el mercado.

La industria del videojuego actualmente goza de una gran vitalidad y presenta una gran variedad de géneros. Sin embargo la evolución del *shoot 'em up* ha llevado a este género al moderno subgénero *bullet hell* (juegos en los que la dificultad se basa única y exclusivamente en disparar y esquivar rápidamente). En este subgénero no importa excesivamente dónde el jugador dispara, sino cómo se mueve para esquivar las balas. Esto demuestra que el *shoot 'em up* es un género de alta dificultad que ha llegado a sus límites en este sentido. ¿Cómo se pueden aportar alicientes a un género que busca la "dificultad por la dificultad" de una manera diferente? Con puzles, en Polarity, sí importa dónde los jugadores disparan y cómo se mueven, más allá de hacerlo para esquivar balas.

Gracias al *post mortem* del videojuego "Sine Mora" [*6, 7*] se extrae que no se le puede dar la espalda al *bullet hell*, por lo tanto en Polarity se opta por rebajarlo y potenciar la dificultad con los puzles y la Inteligencia Artificial (IA) de sus enemigos.

Con este análisis llegamos a la conclusión de que los dos factores que tienen que hacer destacar a Polarity serán los puzles y la IA. Por lo tanto, ya que el desarrollo de puzles


- *E-mail de contacte: Ruben.LozanoG@e-campus.uab.cat*
- *Menció realitzada: Computació*
- *Treball tutoritzat per: Jorge Bernal del Nozal (Ciències de la Computació)*
- *Curs 2014/15*

está ligado a la parte de diseño de videojuegos, se ha decidido centrar el proyecto en el desarrollo de la IA de las entidades inteligentes de Polarity.

## 1.2 OBJETIVOS A ALCANZAR

El proyecto consiste en programar la IA de un videojuego en 3D utilizando la herramienta *Unity3D*. El videojuego necesitará de la implementación de funcionalidades multijugador y ha de ser compatible con periféricos (*gamepads*). Asimismo el proyecto llevará a cabo las tareas de programación de las mecánicas funcionales del juego (maneras que tiene el jugador de interactuar con los elementos del videojuego). Durante el nivel, el jugador podrá interactuar con NPC que estarán dotados de varios niveles de IA; estos niveles de complejidad de IA se verán de forma escalada a lo largo de un nivel, pero también serán accesibles individualmente a través de un menú. El alcance del proyecto es el del desarrollo de la IA hasta un nivel alto de complejidad.
Teniendo en cuenta todo esto, los objetivos a alcanzar durante el desarrollo del proyecto son:

- Diseño de interfaz básica
- Implementación de mecánicas de juego (movimiento del jugador, interacción entre naves, etc…)
- Programación de IA de las entidades enemigas.
- Integración de periféricos para el manejo del juego (ver Figura 1).

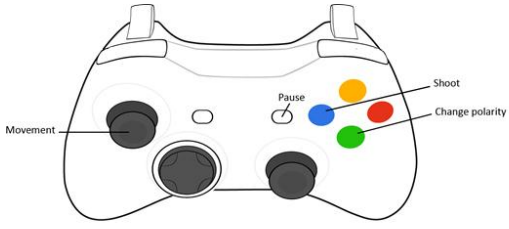



*Figura 1: Esquema de controles de Polarity*

- Diseño de nivel de juego para el *testing* de funcionalidades.

## 1.3 METODOLOGÍA SEGUIDA

La metodología utilizada en este proyecto es la metodología SUM [*8*]. Esta metodología tiene como objetivo la mejora continua del proceso de desarrollo para incrementar su eficacia, en este aspecto se adecua al proyecto ya que cada vez que se añaden módulos se ha de mantener el rendimiento de los mismos. La metodología SUM también nos permite administrar eficientemente los recursos y riesgos del proyecto, logrando así una alta productividad del equipo de desarrollo.

SUM es una adaptación al ámbito de los videojuegos de la estructura y los roles de la metodología *Scrum* [*9, 10*], ya que esta metodología es característica por su flexibilidad para definir el ciclo de vida del proyecto y poder ser combinada fácilmente con otras metodologías de desarrollo. Ello permitiría adaptarse a diferentes situaciones durante el proyecto. El alcance de SUM suele ser para equipos pequeños multidisciplinarios que desarrollen proyectos de corta duración (menos de un año).

## 1.4 REQUISITOS DEL PROYECTO

### 1.4.1 INTERFAZ

La interfaz de Polarity está pensada para que sea lo más minimalista posible y no estorbe al jugador. Por lo tanto se ha decidido que durante el juego, los únicos indicadores que verá el jugador son la vida de las dos naves y un pequeño menú de pausa que se muestra al pulsar la tecla *ESC*.

El menú principal está compuesto de un botón de salida y varios botones que dirigen al jugador a los diferentes niveles de IA individuales. Uno de ellos a un nivel en el que van apareciendo gradualmente todos los módulos de IA desarrollados.

### 1.4.2 REQUISITOS FUNCIONALES

Los requisitos funcionales de Polarity están listados a continuación. Estos requisitos son un compendio de mecánicas de juego que tienen repercusión directa en la jugabilidad:

- Movimiento: Movimiento de las naves en todas direcciones.
- Disparo: La bala del disparo es de la polaridad de la nave que lo ejecuta.
- Recalentamiento: El recalentamiento es una función que penaliza el *spam* de disparo. Ésta mecánica funciona por puntos, aunque estos están ocultos para el jugador.
- Polaridad: La polaridad de la nave se puede cambiar. A las naves protagonistas sólo les afectan las balas de polaridad contraria y a las naves enemigas les afecta cualquier disparo, aunque si son de polaridad contraria les afecta el doble.
- Rayo: El rayo se establece entre dos objetos de polaridad distinta (entre las dos naves protagonistas o entre una de ellas y un objeto especial). Cuando el rayo destruye un enemigo se almacena chatarra que ralentiza a las naves protagonistas. Si ambas naves cambian de polaridad a la vez teniendo el rayo activado (es decir, que el rayo se desactiva y se reactiva en un instante) y tiene chatarra, se produce un efecto propulsor que lanza un rayo hacia adelante (dirección normal al rayo) provocando daño.
- Objetos: En algunos casos se pueden arrastrar objetos mediante el rayo, tirando de ellos.
- Daño:
  - Contacto con rayo: Suma del daño de disparo de ambas naves (si el enemigo que recibe daño tiene polaridad, se multiplica por 2 el daño de la nave de polaridad contraria) + suma daño de la chatarra * factor de tamaño (más tamaño de rayo implica menos daño).
  - Lanzamiento de chatarra: Suma del daño de disparo de las naves destruidas que contenía el rayo * factor de tamaño.
- Muerte: Si una de las naves es destruida, la otra tiene 3 segundos para ir a socorrerla. Dar socorro a la otra nave exige activar el rayo para recargarla y repararla.

3 segundos de socorro harán que la nave destruida vuelva a estar operativa. Si la recarga se interrumpe, por alejarse demasiado o interrumpir el rayo, la nave auxiliadora tendrá otra vez 3 segundos más los segundos que había recargado anteriormente para volver a socorrer.

- *Game Over*: Si una de las naves finalmente muere, el juego muestra una pantalla que te permite reintentar desde el último *checkpoint* o volver al Menú.

### 1.4.3 REQUISITOS DE RENDIMIENTO

En la Tabla 1 se listan los requisitos mínimos y recomendados para poder disfrutar de manera óptima de la experiencia de juego. Aunque los requisitos recomendados son exigentes, Polarity es perfectamente funcional en equipos de gama media.

| | Requisitos mínimos | Requisitos recomendados |
|---|---|---|
| Sistema Operativo | Windows Vista | Windows 7 y superior |
| Memoria RAM | 1 GB | 2 GB y superior |
| CPU | Intel Core a 1.8 GHz o similar | Intel Core Duo a 2.2 GHz o similar |
| Controlador gráfico | DirectX11 | DirectX11 y superior |
| Tarjeta gráfica | 1 GB de RAM dedicada | 1.5 GB de RAM dedicada |
| Espacio en disco | 250 MB | 250 MB |
| Teclado/*Gamepad* | Compatible con controlador XInput | Compatible con controlador XInput |
| Resolución de pantalla | 720p, Aspect ratio 16:9 | 1080p, Aspect ratio 16:9 |

***Tabla 1: Requisitos del sistema***

### 1.4.4 ESCALABILIDAD

Durante el desarrollo del proyecto, se ha previsto en todo momento la escalabilidad de este, desarrollando cada una de las etapas del mismo de forma modular permitiendo de esta manera que se puedan añadir en el futuro más niveles de IA. También se han desarrollado todas las mecánicas y requisitos funcionales, por lo tanto más allá de introducir módulos de IA, se puede escalar en el sentido de añadir niveles de juego y hacer de Polarity un juego más completo en jugabilidad y aventura.

## 1.5 PLANIFICACIÓN INICIAL DEL PROYECTO

### 1.5.1 LISTA DE TAREAS

Fase 1: Requisitos

1. Inicio del proyecto: Generar toda la documentación sobre las definiciones del proyecto necesarias.
2. Planificación: Construir diagrama de Gantt.
3. Análisis: Riesgos y planificación de plan de contingencia.
4. Requisitos: Análisis de requisitos (funcionales, rendimiento e interfaz) del proyecto.

Fase 2: Generación versión *alpha*

5. Programación de controladores: Integración de periféricos
6. Extracción de mecánicas del *Game Design Document* (GDD): Listado de requisitos funcionales a implementar.
7. Implementación de mecánicas
8. Creación de primer prototipo *alpha*: Prototipo que permite jugar a Polarity sin enemigos con todas sus mecánicas.

Fase 3: Desarrollo IA

9. Generación de módulos de IA

Fase 4: Generación versión *beta*

10. Integración gráfica: Integración de modelos de enemigos y naves.
11. Maquetación del nivel
12. Prototipo *beta*: Acceso por separado a demos de todos los niveles de IA y a un nivel que agrupa todos los módulos de forma escalada.
13. Período de pruebas: Distribución de *beta* y obtención de *feedback*.

Fase 5: Final

14. Prototipo final
15. Documentación
16. Entrega del producto

### 1.5.2 RECURSOS DISPONIBLES

#### 1.5.2.1 EQUIPO TÉCNICO

- Rubén Lozano Gómez: Encargado de implementar la parte técnica y funcional del proyecto.

#### 1.5.2.2 EQUIPAMIENTO

- PC de sobremesa: 1 PC de sobremesa con todo el material necesario para la correcta implementación de los *milestones* del proyecto. También se puede utilizar como terminal de prueba en las diferentes fases de *testing*.
- Gamepad: 2 *gamepads* con los controladores necesarios que se especifican en los requisitos para poder hacer pruebas en la fase de implementación de periféricos.

#### 1.5.2.3 HERRAMIENTAS

- *Unity3D*: La herramienta a utilizar durante todo el proyecto como ya se ha mencionado es el *engine Unity3D* que permite el desarrollo del videojuego y todos sus módulos durante todo el ciclo de vida del proyecto.

### 1.5.3 RIESGOS DEL PROYECTO

| RIESGO | PROBABILIDAD | IMPACTO | DESCRIPCIÓN | SOLUCIÓN |
|---|---|---|---|---|
| Parte del cronograma de actividades supera el tiempo fijado | Media | Alto | Todas las actividades que sean dependientes de la que se ha pasado en tiempo se verán afectadas a modo de retraso | Crear un cronograma con un colchón de tiempo antes de cada fin de fase, referenciado en el Gantt como "Final de fase" |
| Problema con el equipamiento o herramientas | Baja | Alto | Todo el proyecto queda pausado hasta que se disponga de nuevos equipos o herramientas | Buscar nuevo equipamiento y buscar en foros |

***Tabla 2: Riesgos principales de desarrollo***

Los riesgos principales del proyecto (ver Tabla 2) podrían venir por un posible error en la planificación del mismo que supondría un retraso en la realización de las tareas. Otro posible riesgo sería la indisponibilidad de herramientas de trabajo, lo cual pararía el desarrollo del mismo. Riesgos menores serían la falta de disponibilidad del arte del videojuego, aunque en este caso no afectaría ni al desarrollo ni al resultado de este proyecto.

### 1.6 ESTRUCTURA DEL DOCUMENTO

La estructura del artículo es la siguiente: en esta Sección se han definido los objetivos del proyecto y requisitos recogidos para el desarrollo del mismo. La Sección 2 es el estado del arte, donde se hace una breve introducción histórica a los videojuegos, a las herramientas de desarrollo de éstos con especial mención a las técnicas de IA. La Sección 3 explica cómo se ha desarrollado el proyecto, incluyendo diagramas de comportamiento. En la Sección 4 se expondrán contenidos de las versiones producidas y se analizará la planificación seguida y los problemas encontrados. El documento finaliza en la Sección 5 detallando el estado actual del juego con un apartado de conclusiones donde se reflexiona sobre los conocimientos adquiridos.

## 2 ESTADO DEL ARTE

### 2.1 INTRODUCCIÓN

Para el desarrollo de videojuegos se utilizan *game engines* (motores de videojuegos). Cuando se habla de *game engine* se hace referencia a un software diseñado para la creación y el desarrollo de videojuegos, este proceso engloba el diseño, la creación y la representación de un videojuego.

La funcionalidad que todo motor de videojuegos debe tener es la de proveer al videojuego un motor de *rendering*, un sistema de físicas, gestión de sonido, *scripting* y administración de memoria. La principal ventaja de usar un motor es que la mayoría de motores permiten una exportación multiplataforma y proveen de herramientas de desarrollo visual así como componentes de software optimizado en forma de *Integrated Development Enviroment* (IDE) [*11*].

En el mercado se pueden encontrar múltiples motores de videojuegos, de los cuales se ha elegido *Unity3D*. Se ha tomado esta decisión debido a la flexibilidad del motor en cuanto a estética de juego y género, a la experiencia en el mismo, y al familiarización con el lenguaje de programación que utiliza (C#). Todas estas razones favorecen la implementación de un código óptimo y un desarrollo ligero, que son dos de los principales requisitos que tiene este proyecto. Debido al nivel de la complejidad de la IA es importante que todo el código relacionado con mecánicas de juego y control de eventos consuma el mínimo coste de proceso posible, y la limitación de tiempo implica un desarrollo rápido de todos los componentes.

### 2.2 IA EN LOS VIDEOJUEGOS

#### 2.2.1 INTRODUCCIÓN

La IA se divide en dos grupos, los cuales aplican técnicas muy diferentes:

- Determinista: Conjunto de técnicas con un comportamiento especificado previamente al tiempo de ejecución, tendiendo así a ser predecibles. Son las técnicas más utilizadas.
- No Determinista: Conjunto de técnicas con un grado de incertidumbre en cuanto al resultado de las acciones a realizar. Este grado de incertidumbre depende de la técnica concreta utilizada y su aplicación. Este grado de incertidumbre viene dado por el proceso de aprendizaje que hace la entidad, que puede ser diferente en cada situación. Estas son técnicas menos utilizadas debido a su complejidad y su difícil *testing*.

En el campo de los videojuegos se utilizan tanto técnicas deterministas como no deterministas, y el compendio general de todas estas técnicas son las enumeradas a continuación [*12*].

#### 2.2.2 PATRONES DE MOVIMIENTO

El objetivo de las técnicas de patrones de movimiento es determinar la posición final de una entidad. En la mayoría de casos las soluciones a los planteamientos son asumibles, la complejidad del problema surge si se ha de encontrar la mejor ruta para la posición final, que se tratará en la siguiente técnica.
Los planteamientos que surgen de estas técnicas son los siguientes:

- Movimiento aleatorio: Se dispone de un conjunto de puntos y aleatoriamente se escoge uno como objetivo.
- Patrones: Hay un punto inicial y final definidos desde el inicio. Estos no varían.
- Movimiento probabilístico: Se dispone de un conjunto de posiciones, las cuales tienen un peso asociado. Este peso influye en la probabilidad que tienen de ser elegidas como posición objetivo.
- *Flocking*: La entidad a trasladar de una posición a otra es un grupo de entidades, este planteamiento le da a ese grupo propiedades especiales (pequeñas desviaciones, separaciones…). Una posible implementación se encuentra en [*13*].

#### 2.2.3 BÚSQUEDA DE CAMINOS

El objetivo de las técnicas de búsqueda de caminos es determinar la ruta que debe seguir un objeto desde su posición de origen hasta la posición final teniendo en cuenta múltiples factores (terreno, obstáculos…). El sistema de representación utilizado son grafos dirigidos.
Los planteamientos que surgen de estas técnicas son los siguientes:

- Caminos predefinidos: La entidad dispone de posiciones definidas que marcan su ruta.
- Exploración sin coste: Se explora el grafo hasta encontrar una solución válida. Implementaciones de este algoritmo incluyen métodos de exploración por profundidad, exploración por anchura o algoritmos

*greedy* [*14*].
- Exploración con coste: Exploración del grafo hasta encontrar una solución válida con coste mínimo. Algoritmos como A* [*15*], *Dijkstra* [*16*], *Backtracking* [*17*], Programación dinámica [*18*] o Mallas de navegación [*19*] ofrecen soluciones válidas para esta técnica.

#### 2.2.4 SIMULACIÓN DE SISTEMAS

El objetivo de las técnicas de simulación de sistemas es emular el comportamiento de un objeto. El sistema de representación utilizado es el Árbol de decisiones.
Los planteamientos que surgen de estas técnicas son los siguientes:
- "Hacer trampas": Se tiene acceso a toda la información del oponente. Se ha de balancear el poder para el control de la frustración del jugador. Ejemplos de estos sistemas son: *Minimax* [*20*], Poda *alfa-beta* [*21*], *Backtracking*.
- Sistema de reglas: Se dispone de una serie de reglas simples con condiciones, que al ser cumplidas activaran ciertos comportamientos
- Estados y transiciones: Diagramas de estados y transiciones que determinan el comportamiento de un agente. Sistemas como *Finite State Machines* (FSM) [*22*] y Árboles de comportamiento [*23*] son ejemplos de implementación.

#### 2.2.5 APRENDIZAJE

El objetivo de las técnicas de aprendizaje es modificar el comportamiento de una entidad durante el tiempo a través del conocimiento adquirido.
Los planteamientos que surgen de estas técnicas son las siguientes:
- Aprendizaje por refuerzo: Cada acción realizada tiene su posterior valoración (positivamente o negativamente) respecto a sus consecuencias, por lo tanto estas valoraciones serán tomadas en cuenta cuando se vayan a ejecutar futuras acciones. Un ejemplo de este sistema se encuentra en [*24*].
- Aprendizaje deductivo: A partir del conocimiento se deduce nuevo conocimiento, creando así bases de conocimiento estructuradas, de la misma forma que en Redes bayesianas [*25*].

#### 2.2.6 PLANIFICACIÓN

El objetivo de las técnicas de planificación es el cálculo de una serie de acciones que una vez hechas darán como resultado nuestro objetivo.
El planteamiento que surge de estas técnicas es el siguiente:
- Agentes guiados por metas: Cada agente dispone de información sobre el mundo, con la cual son capaces de construir un plan basado en acciones con coste asociado, pasando de esta manera de estado actual al estado objetivo. Se puede encontrar un ejemplo en [*26*].

#### 2.2.7 LÓGICA DIFUSA

El objetivo de las técnicas de lógica difusa es dotar a una máquina de estados con una condición de cumplimiento ambigua (entidad enemiga cerca del jugador, poca vida…).

#### 2.2.8 REDES NEURONALES

El objetivo de las técnicas de redes neuronales es simular el comportamiento de neuronas humanas. A partir de unos datos de entrada tras la aplicación a éstos de funciones de cómputo y activación a través de varias capas se genera una salida.
El planteamiento que surge de estas técnicas es el siguiente:
- Aprendizaje RNA: Modificación del peso de las conexiones entre entradas y funciones para producir ciertos patrones deseados, tal y como se observa en algoritmos de redes recurrentes [*27*]

## 3 DESARROLLO DEL PROYECTO

Para el desarrollo del proyecto, más allá de la implementación de las mecánicas de juego -que han tenido un gran componente de complejidad- se ha tenido en cuenta desde el principio que el objetivo principal de desarrollo era la programación de la IA. Respecto a esto, se ha querido llegar a un nivel de complejidad de IA alto. Para lograrlo en algunos casos se ha desarrollado más de un nivel de dificultad de una misma técnica para poder explorar con mayor profundidad lo que esta técnica puede brindar. A continuación se expondrán de manera detallada los módulos de IA que se han implementado.

### 3.1 NIVEL DE COMPLEJIDAD BAJO

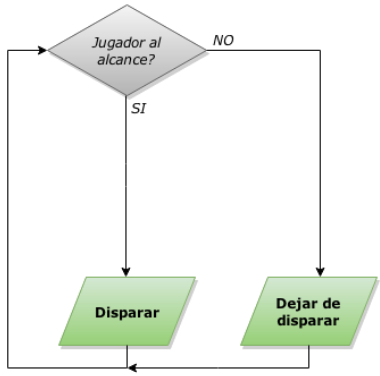
#### 3.1.1 PATRONES (PATRONES DE MOVIMIENTO)

Definición de una serie de posiciones espaciales por las cuales se desplazará una entidad. En este caso sólo se trata de movimiento, aunque la entidad en este caso sí podrá atacar no tendrá ningún comportamiento especial, ya que estos casos se tratarán en FSM, que también permitirán patrones de movimiento.
La implementación de estos patrones, han supuesto una programación orientada a la configuración de las rutas de las entidades con opciones modificables desde el editores tales como: ping-pong, bucle, aleatoriedad, etc… Estas opciones permiten la personalización de cada ruta. Cada entidad tiene asignado su camino a seguir.

#### 3.1.2 LÓGICA DIFUSA

Metodología que se basa en definir acciones que se activarán si ciertas condiciones (nave cerca, poca vida, etc…) se cumplen. Es la manera más simplificada de hacer una FSM, ya que solo dependerá de esa condición, para pasar a una acción concreta. Se han desarrollado dos módulos de lógica difusa, para que cada enemigo pueda tener diversos niveles de dificultad en esta técnica; cada nivel hereda del anterior nivel y todos ellos implementan las funcionalidades de patrones simples (ver Figura 2 y Figura 3).

#### 3.1.2.1 PRIMER NIVEL




#### 3.1.2.1 SEGUNDO NIVEL

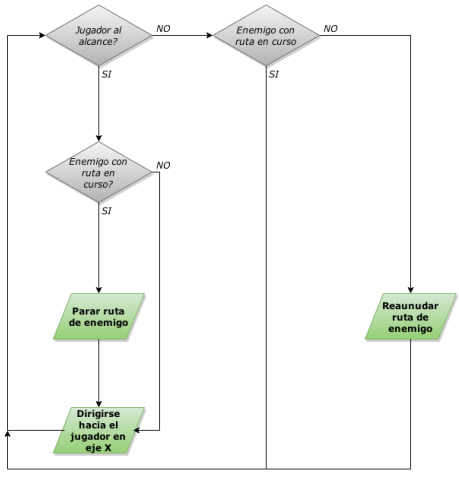



*Figura 3: Diagrama Lógica Difusa 2*

## 3.2 NIVEL DE COMPLEJIDAD MEDIO

### 3.2.1 FSM (ESTADOS Y TRANSICIONES – SIMULACIÓN DE SISTEMAS)

La técnica de FSM permite relacionar estados con transiciones de una manera más compleja que con la lógica difusa, otorgando a la entidad un comportamiento más autónomo, dependiendo de cómo se relacionen los estados con estas transiciones. Por lo tanto, la implementación de FSM requiere de un conjunto de estados posibles que puede tomar una entidad, estos estados están supeditados a un conjunto de transiciones, las cuales orquestan el comportamiento final de la entidad.

Cada transición tiene unas condiciones y cada estado unas acciones, siendo esta combinación "condición-estado" la que hace que la entidad tenga un comportamiento inteligente *[28, 29, 30]*.

La implementación de FSM ha sido realizada utilizando una metodología de llamadas por estado, es decir, cada estado tiene predefinidas dos llamadas: *Start* y *Update*, lo que permite codificar todo lo que se necesite en estas dos llamadas que serán invocadas al iniciar al estado y en cada ciclo de reloj respectivamente. La gestión de los estados se hace mediante una llamada al sistema que tiene acceso a la definición de todos lo estados de una FSM que permite pasar de un estado a otro gracias a esta definición estática. Por lo tanto, gracias a esta metodología desarrollada, la escalabilidad de FSM es tan sencilla como implementar una máquina de estados que herede de las clases correspondientes e implementar nuestros nuevos estados personalizados, cada uno con una funcionalidad y una gestión propia. Es decir, al implementar una máquina de estados nueva, la única complejidad es la lógica interna de cada estado y no la gestión de la misma.

Para una mayor facilidad de implementación antes de desarrollar un nivel de inteligencia más complejo de FSM se ha seguido la metodología de diseñar un diagrama de grafos de las transiciones de estado previo, haciendo que sea más fácil de implementar y hacer el *testing* de comportamiento de cada entidad.

A continuación se exponen los tres niveles de FSM que han sido desarrollados.

#### 3.2.1.1 PRIMER NIVEL

Es el nivel más básico de FSM, se ha incluido para demostrar la implementación de estados y la posibilidad de añadir acciones a la entidad relacionadas con el estado. Asimismo se ha implementado el uso de eventos por rango. En este caso este tipo de eventos se refiere a que la acción a ejecutar depende de que la entidad jugador entre en un rango de distancias respecto de la entidad inteligente (ver Figura 4).

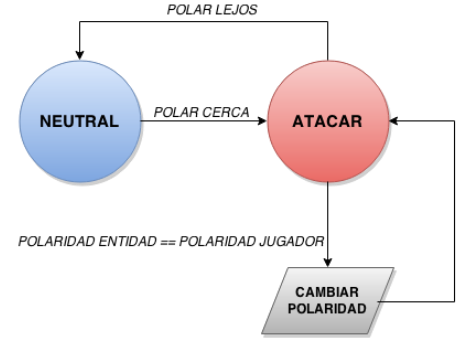



*Figura 4: Diagrama FSM 1*

#### 3.2.1.2 SEGUNDO NIVEL

En este nivel se ha querido hacer más compleja la idea de los ciclos FSM, queriendo demostrar en este nivel que se pueden dar rupturas de ciclo teniendo estados a los que ya no se podrá regresar. En este caso concreto podemos observar que una vez la entidad esté en el estado de atacar ya no volverá nunca al estado neutral (ver Figura 5).

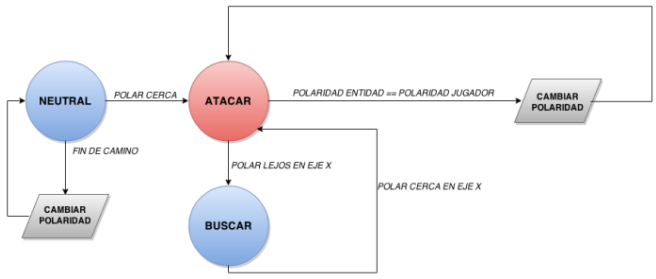



*Figura 5: Diagrama FSM 2*

#### 3.2.1.3 TERCER NIVEL

En este último caso se pretende demostrar que con la metodología FSM, aunque un estado sea complejo en el sentido de la lógica que requieren sus acciones, esto no supone ningún tipo de complejidad añadida más allá de la implementación de la propia lógica. Esto se puede observar con el estado de huir que tiene sus propias reglas dentro del propio FSM (ver Figura 6 y Figura 7).

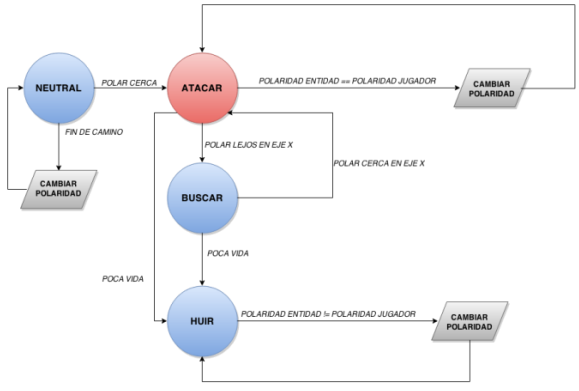



*Figura 6: Diagrama FSM 3.1*

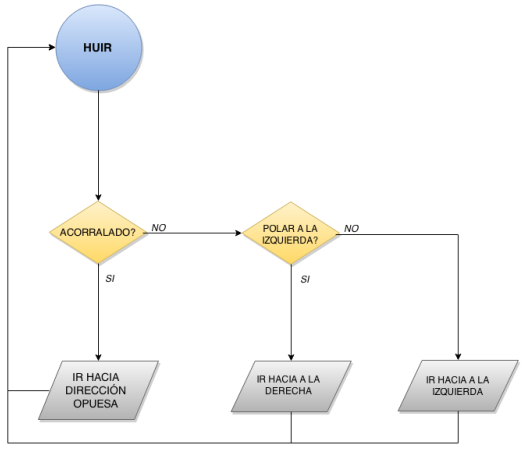



*Figura 7: Diagrama FSM 3.2*

## 3.3 NIVEL DE COMPLEJIDAD ALTO

### 3.3.1 *FLOCKING* (PATRONES DE MOVIMIENTO)

La técnica de *Flocking* permite dar un comportamiento especial a un grupo de entidades que se mueven hacia un objetivo aplicando el algoritmo *Boids* de Craig Reynlods. Este algoritmo consiste en que un grupo de entidades en movimiento eviten el movimiento uniforme: para conseguir este efecto se aplica un modelo de comportamiento que tiene su base en el comportamiento que tienen las aves al desplazarse en grupos *[31, 32, 33]*.

El algoritmo está dividido en tres etapas que se ejecutan secuencialmente cada ciclo de reloj por cada entidad que forma parte del grupo de entidades que se mueven:

1. **Separación**: Durante esta etapa cada entidad tiene definido un radio de distancia en el cual no puede haber ninguna otra entidad. Por tanto, se calcula si hay alguna entidad dentro de ese radio y se obtiene la suma vectorial de los vectores de posición de estas entidades respecto a la que está ejecutando el algoritmo. El resultado es un vector normalizado inverso que hace que la entidad se separe de estas entidades en posiciones demasiado cercanas (ver Figura 8).

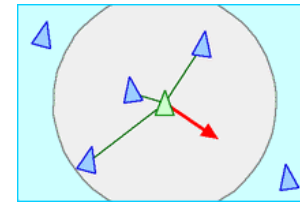


*Figura 8: Ejemplo de etapa de separación en Flocking [34]*

2. **Cohesión**: El algoritmo de *Flocking* impone que todas las entidades tienen que estar en una posición cercana al centro de gravedad del grupo. Por lo tanto, para cada entidad se calcula la posición media de las otras entidades, dando como resultado un vector normalizado de ese centro de gravedad calculado respecto a la entidad que está ejecutando el algoritmo (ver Figura 9).

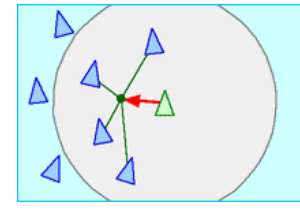


*Figura 9: Ejemplo de etapa de cohesión de Flocking [34]*

3. **Alineamiento:** Para finalizar, con el fin de conseguir el efecto de que todas las entidades se desplacen a igual velocidad, se calcula un nuevo vector normalizado de la media de la velocidades de las demás entidades (ver Figura 10).

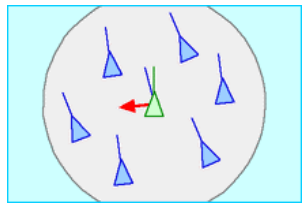


*Figura 10: Etapa de alineamiento de Flocking [34]*

Una vez se calculan estos tres componentes vectoriales, la suma de éstos dará como resultado el vector director final de nuestra entidad. Para que el algoritmo sea más personalizable cada componente tiene un peso asignado para que se pueda modificar el comportamiento de entidades que queremos diseñar con este algoritmo.

A parte del algoritmo original explicado anteriormente, el módulo de IA tiene algunos añadidos:

- Componente de ruido: Se genera un vector de ruido a cada entidad para disminuir el efecto simétrico en el que parece que todas las entidades siguen un mismo patrón de movimiento.
- Objetivo común: Se genera en este caso otro vector director que apunta directamente al punto objetivo, obligando así a todas las entidades a mantener un mismo objetivo.

Las dos componentes añadidas también tienen un peso asignado que puede ser modificado para incrementar el impacto de una característica determinada.
El algoritmo está diseñado de forma que desde el editor podemos modificar el radio de efecto de cohesión y separación, así como los pesos de cada componente, con el objetivo de una configuración fácil e intuitiva.

### 3.3.2 ÁRBOLES DE COMPORTAMIENTO (SIMULACIÓN DE SISTEMAS)

Método que presenta una *interface* de editor exactamente igual a la de FSM pero con las opciones especiales del modelo de tareas de los árboles de comportamiento *[35, 36, 37, 38]*. Interiormente se comporta de manera diferente a la FSM ya que aplica la metodología propia de éstos árboles (Tarea, Selector, Secuencia y *Parallel*) *[39]*.
Cada componente del árbol de decisión está clasificado dentro de una de las siguientes categorías:

- Tarea: Acción o condición (Representada con una caja)
- Secuencia: Conjunto de tareas (Representada en forma de flecha)
- Selector: Nodo padre de tareas o secuencias (Representada con un interrogante)
- *Parallel*: Padre de tareas que ejecuta a sus tareas hijas en paralelo (Representada con una caja de doble línea)

La metodología que se utiliza es entrar en el selector, explorar la primera rama de secuencia y en caso de que la secuencia tenga éxito (todas las condiciones han devuelto un resultado positivo), se vuelve a explorar el árbol desde su raíz (el selector). En caso de que un selector no tenga éxito (una condición devuelve un resultado negativo) se pasa a la siguiente secuencia hija que esté a la derecha de la última explorada (ver Figura 11).

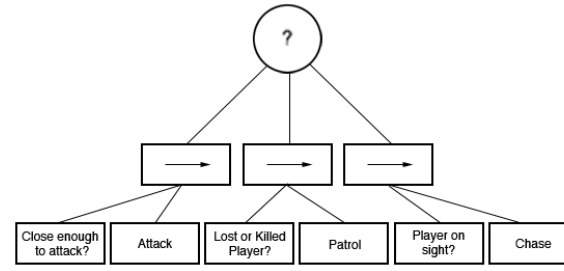



*Figura 11: Ejemplo de árbol de comportamiento [14]*

La estructura de árboles nos permite que la toma de decisiones de la entidad obtenga resultados muy similares a los conseguidos con FSM. La particularidad de los árboles de decisión reside en que teniendo en cuenta su modelo de procesamiento de acciones y condiciones, permite incrementar mucho el nivel de complejidad de la entidad inteligente, reduciendo así tanto el coste de implementación como de ejecución que supondría hacerlo en FSM en entorno *Unity3D*. Esto es posible gracias a las instrucciones paralelas que puede explotar el *multi-threading* de *Unity3D* que reduce mucho la ejecución de gran parte de código gracias a su estructura de salto.
Para la implementación de los árboles de comportamiento se ha seguido la metodología definida en la Figura 12.

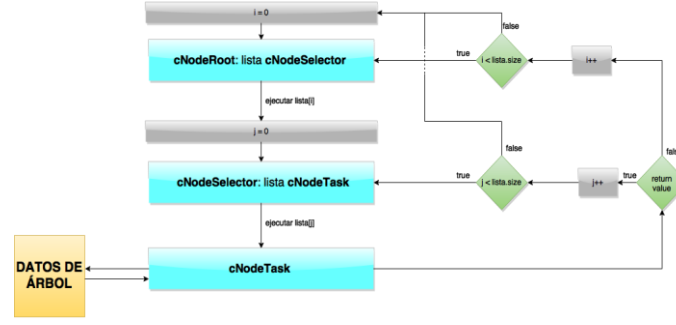



*Figura 12: Diagrama de flujo de árboles de comportamiento*

Como podemos observar en el diagrama, tenemos los siguientes elementos:

- Lista de selectores: Representación de los nodos hijos de la raíz del árbol
- Lista de tareas: Representación de los nodos hijos de cada selector del árbol
- Datos de árbol: Cada tarea tiene una referencia a un módulo de datos que tiene funciones y atributos propios de la lógica de cada árbol. Por lo tanto, cada árbol siempre ha de tener un módulo de datos para que las tareas de éste puedan acceder a esta información.

Por lo tanto podemos concluir en que el árbol siempre dispondrá de tres capas (en orden descendiente): Raíz, Selector y Tarea.
El funcionamiento del árbol es el siguiente:

1. Se ejecuta el primer elemento de la lista de selectores (Capa de Raíz a Capa de Selector).

2. Esto implica que se ha de ejecutar el primer elemento de la lista de tareas que tiene dicho selector (Capa de Selector a Capa de Tarea).
3. La tarea se ejecuta y actualiza si es necesario los atributos del módulo de datos del árbol (Capa de Tarea).
4. La ejecución de la tarea siempre devolverá un valor (verdadero/falso). Al estar implementado de una manera recursiva este valor será devuelto a la capa de arriba (dependiendo del resultado a una capa u otra 5.1/5.2/5.3).

5.1 En caso positivo se ejecuta la siguiente tarea y volvemos al punto 4 (Capa de Tarea a Capa de Selector).

5.2 En caso negativo se ejecuta el siguiente elemento de la lista de selectores y se vuelve al punto 2 con el nuevo selector como objetivo (Capa de Tarea a Capa Selector)

5.3 En el caso de que o bien se hayan ejecutado todas las tareas o todos los selectores, se vuelve al punto 1 (Capa de Tarea a Capa de Raíz).

Esta implementación en *Unity3D* ha tenido una complejidad añadida. Esto se debe a que se han tenido que controlar los ciclos de reloj de la función *Update* nativa, ya que al tener que ejecutar ésta función recursiva en forma de bucle, se han tenido que limitar las ejecuciones por ciclo de reloj.

El desarrollo de esta técnica se ha llevado a cabo a través de una metodología basada en herencias, haciendo que su nivel de escalabilidad sea muy alto. De esta manera se permite crear un nivel de comportamiento tan solo implementando el módulo de datos de árbol con la lógica necesaria para el árbol y extendiendo la clase *cNodeTarea* con su funcionalidad propia.

Finalmente se ha diseñado un diagrama de comportamiento para esta técnica (ver Figura 13).

#### 3.3.2.1 DIAGRAMA DE COMPORTAMIENTO

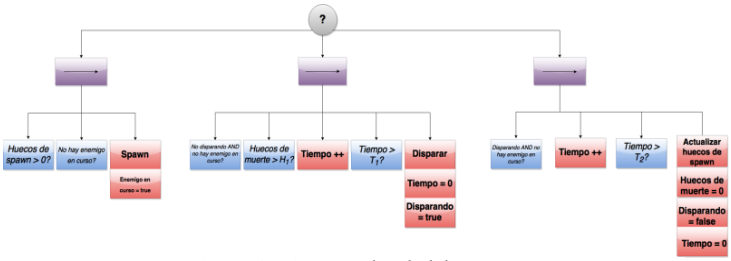



*Figura 13: Diagrama de árbol de comportamiento*

Para este árbol de comportamiento existen diversas variables modificables:

- $T_1$: Tiempo de carga
- $T_2$: Tiempo de reposo
- $H_1$: Número de enemigos a eliminar para que la entidad empiece a disparar.

El funcionamiento del árbol se basa en la creación de enemigos pequeños que se ponen enfrente de un enemigo mayor haciendo de barrera. Una vez el jugador elimina cierto número de enemigos, la entidad espera un tiempo de carga y dispara una potente ráfaga, posteriormente espera el tiempo de reposo para volver a crear enemigos que le sirven de escudo y así sucesivamente.

Como se puede ver, este comportamiento se basa en observar la cantidad de enemigos eliminados por los jugadores y utilizar una serie de tiempos para poder disparar.

## 4 RESULTADOS

Los resultados del proyecto se pueden mostrar a lo largo de sus dos versiones: *alpha* y *beta*. Además de estas versiones, durante el desarrollo del proyecto se han producido *builds* de contenido reducido (módulos de IA, mecánicas, etc…) para poder hacer el *testing* de funcionalidades de una forma más ágil.

En la versión *alpha* se puede ver cómo están implementadas todas las mecánicas. Para llegar a esta versión se tuvo que someter la versión a un período de prueba para poder asegurar su calidad, y que cada mecánica era funcionalmente estable. Para esta versión éstas son las mecánicas que fueron implementadas:

- Configuración de vista de cámara (ver Figura 14)
- Movimiento de las naves con el teclado y *gamepad*
- Restricciones de colisiones con el entorno
- Gestión de vida de la naves protagonistas y *Game Over*
- Disparo con partículas, daño y sobrecalentamiento.
- Todas las mecánicas que tienen que ver con el rayo (entre protagonistas, entre objetos especiales, etc…)
- Rescate de nave compañera con rayo
- Diseños de *interfaces* de usuario

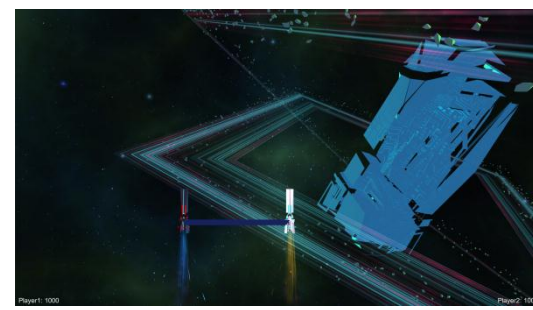



*Figura 14: Escena de juego*

La versión *beta* incluye todas las funcionalidades de la versión *alpha* pero con el añadido de todos los módulos de IA. Al contrario que la versión anterior, esta versión sí que incluye un menú principal donde podemos elegir entre la versión de IA que deseada. Dada la complejidad que supone mostrar gráficamente lo que supone esta versión *beta* estará disponible un enlace donde poder descargarla y para probarla en Windows. El contenido

adicional respecto a la versión *alpha* que incluye esta versión es el siguiente:

- Patrones
- Lógica difusa Nivel 1
- Lógica difusa Nivel 2
- FSM Nivel 1
- FSM Nivel 2
- FSM Nivel 3
- *Flocking*
- Árbol de comportamiento
- Nivel de Prueba

### 4.2. REVISIÓN DE LA PLANIFICACIÓN

En la planificación inicial había una tarea que al final no se ha desarrollado, y es la FSM Jerárquicas. Se tomó esta decisión porque se consideró que no aportaba suficiente innovación a nivel de complejidad, ya que se parecía mucho a las FSM. En cambio se optó por desarrollar más niveles de complejidad en las IA de lógica difusa y FSM. De esta forma se pudo observar la variación en la escala de dificultad que puede aportar una sola técnica por si sola. Todas lo demás tareas se han desarrollado según lo planeado, sin más inconvenientes que los ya planificados producidos por errores detectados en las fases de *testing*.

### 4.3. PROBLEMAS ENCONTRADOS

Uno de los problemas encontrados fue que durante el desarrollo del proyecto hubo una actualización del *game engine* de *Unity3D*, y hubo un proceso de actualización del proyecto para poder trabajar siempre en la última versión y poder utilizar las mejoras introducidas en las versiones más recientes.

Otro problema fue que debido a la complejidad de la integración gráfica, no se pudo llegar al nivel de artístico que se esperaba. Aunque a nivel de programación este problema no se puede resolver, se puede estimar que al ser un proyecto ambicioso, con más tiempo el nivel de gráfico de Polarity podría haber sido superior.

## 5 CONCLUSIONES

### 5.1. OBJETIVOS DE APRENDIZAJE ADQUIRIDOS

El principal objetivo de este proyecto era el de poder llevar a cabo un proyecto de larga duración con una hoja de ruta bien definida, pudiendo enfrentar los contratiempos surgidos en el ámbito de la ingeniería informática. Dicho objetivo se ha cumplido con creces, ya que incluso se han podido añadir algunas modificaciones a la planificación inicial, sin que el proyecto haya sufrido en calidad. Otro de los objetivos era el de poder diseñar una IA con un propósito concreto a través de un estudio del estado del arte de la IA en este campo concreto, y a pesar de que no se ha podido profundizar mucho en cada técnica, se puede decir que la IA implementada es ideal para este género y se podrá aprovechar para proyectos futuros. Y el último objetivo era el de ser capaz de generar documentación de proyectos informáticos siguiendo los estándares establecidos, que a lo largo de los informes de seguimiento y de planificación se ha podido ir mejorando iterativamente gracias a la supervisión del tutor de este proyecto.

### 5.2. LÍNEAS FUTURAS

Polarity es un proyecto muy ambicioso de larga duración, este TFG solo ha sido una pequeña parte de lo que realmente supone Polarity. Gracias a este proyecto se ha podido desarrollar una IA de calidad para un videojuego que no acaba aquí. A partir de la versión *beta* se ha establecido una base en la que poder empezar a implementar niveles que cuyo objetivo sea la diversión y no la complejidad de las entidades inteligentes. Para dicho cometido es preciso el rol de un *game designer* que se dedique única y exclusivamente a esta tarea y darle a Polarity este factor que no se puede conseguir en un proyecto de computación, pues los objetivos no son los mismos.

El próximo paso para Polarity es reunir un equipo formado por los compañeros que me han ayudado externamente a nivel artístico y de diseño, y planificar un proyecto con gran futuro como es éste y convertir este proyecto en un videojuego de éxito en todos los sentidos.

## AGRADECIMIENTOS

En primer lugar me gustaría agradecer a mi tutor Jorge Bernal, por toda la ayuda que me ha ofrecido durante el proyecto y por estar tan ilusionado como yo desde que junto acabamos de pulir la idea de Polarity en la fase inicial del proyecto. Me han servido mucho todas las reuniones semanales y las correcciones a todos los documentos y versiones, para poder dirigir este proyecto hacia el punto que quería.

En segundo lugar me gustaría agradecer todo el trabajo realizado por mis compañeros Dani y David, que han aportado a este proyecto de ingeniería informática, una componente visual y artística de una calidad muy superior a la que he podido realizar anteriormente. Gracias al diseño aportando se ha conseguido un nivel de trasfondo y jugabilidad que hacen de este proyecto algo muy especial y único.

Por último agradecer a amigos y familiares todo el apoyo e ilusión que me han mostrado durante el transcurso de este difícil proyecto, que ha resultado ser un gran reto a nivel personal, sin ellos se me habría hecho mucho más complicado afrontar Polarity de la manera en que lo he hecho.

## GLOSARIO

*Build*: Ejecutable generado a partir de la escena de *Unity3D*.

*Checkpoint*: Punto de reaparición después de morir durante partida.

*Game Design Document*: Documento en el que se recogen todas las decisiones de diseño.

*Gamepad*: Periférico especial para uso exclusivo en videojuegos.

*Interface*: Capa de interacción intermedia que permite al usuario la navegación a través de los menús.

*Multi-threading*: Ejecución de simultánea de diferentes secciones de código.

*Post mortem*: Análisis y conclusiones extraídas de un proyecto ya terminado.
*Rendering*: Proceso de generar un conjunto de imágenes a partir de un modelo 3D.
*Scripting*: Metodología que consiste en organizar el comportamiento de los objetos mediante scripts independientes.
*Shot-em-up*: Género de videojuegos en los que el jugador controla un personaje que dispara contra oleadas de enemigos que van apareciendo.
*Spam*: Secuencia continuada de una misma acción a mucha velocidad.
*Testing*: Conjunto de pruebas que se realizan de un código concreto con el fin de encontrar errores.
*Unity3D*: Motor gráfico de creación de videojuegos multiplataforma.

## BIBLIOGRAFÍA


[1] Grodal, T. (2000). *Video games and the pleasures of control. Media entertainment: The psychology of its appeal*, pp. 197-213.
[2] Varios autores, "Radiant Silvergun", http://goo.gl/aCM1FY, Última visita: 18/06/2015
[3] Varios autores, "Ikaruga", http://goo.gl/Uc4jIh, Última visita: 18/06/2015
[4] Varios autores, "Geometry Wars": Retro Evolved 2, http://goo.gl/aE7g3x, Última visita: 18/06/2015
[5] Varios autores, "Resogun", http://goo.gl/py79y5, Última visita: 18/06/2015
[6] Varios autores, "Sine Mora", http://goo.gl/6sKM3p, Última visita: 18/06/2015
[7] Varios autores, *Post mortem* de "Sine Mora", http://goo.gl/05Yz4m, Última visita: 18/06/2015
[8] Varios autores, Metodología SUM, http://goo.gl/hCuXh3, Última visita: 18/06/2015
[9] Varios autores, Metodología *Scrum*, http://goo.gl/CSVBjB, Última visita 18/06/2015
[10] Varios autores, Metodología *Scrum*, http://goo.gl/pxpTxm, Última visita 18/06/2015
[11] Varios autores, Motor de videojuegos, http://goo.gl/yeoe1N, Última vista: 18/06/2015.
[12] Juan Gómez Romero, Clasificación inspirada en *Introducción a la Programación Lúdica*, http://goo.gl/fQudhR, Última visita: 18/06/2015.
[13] Craig Reynlods (1987). *Flocks, Herds and Schools – A Distributed Behavioral Model*.
[14] Hezewinkel Michel (2001). *Greedy algorithm*.
[15] Nilsson, N.J (1980). *Principles of Artificial Intelligence*.
[16] Corme, Thomas H., Leiserson Charles E., Rivest Ronald L., Stein, Clifford (2001). *Introduction to Algorithms*.
[17] Eitan Gurari (1999). *Backtracking algorithms*.
[18] Richard Bellman (2003). *Dynamic Programming*
[19] Sandy Brand (2009). *Efficient obstacle avoidance using autonomously generated navigation meshes*.
[20] Russel, Stuart J., Norving, Peter (2003). *Artificial Intelligence: A modern approach*.
[21] Richard D. J, Hart T. P., (1963). *The alpha-beta heuristic*.
[22] Wright, David R. (2005). *Finite State Machines*.
[23] Boston J. (2008). *Behaviour Trees – How they improve engineering behavior?*
[24] S. Cagnoni (2000). *Real-world applications of evolutionary computing*.
[25] Neapolitan, Richard E. (2004). *Learning Bayesian networks*.
[26] David E. Wilkins (2000). *SIPE-2: System for interactive planning and execution*.
[27] Rául Rojas (1996). *Neural networks: a systematic introduction*.
[28] Cheng, K. T., & Krishnakumar, A. S. (1993, July). *Automatic functional test generation using the extended finite state machine model. In Proceedings of the 30th international Design Automation Conference* (pp. 86-91). ACM.
[29] Bosik, B. S., & Uyar, M. Ü. (1991). *Finite state machine based formal methods in protocol conformance testing: from theory to implementation. Computer Networks and ISDN Systems*, 22(1), 7-33.
[30] Ejemplo de videojuego que utilzia FSM para controlar a los enemigos, http://goo.gl/IPOFsl, Última visita: 18/06/2015.
[31] Reynolds, C. W. (1999, March). *Steering behaviors for autonomous characters*. (Vol. 1999, pp. 763-782).
[32] Lee, J. M. (2010). *An efficient algorithm to find k-nearest neighbors in flocking behavior. Information Processing Letters*, 110(14), 576-579.
[33] Ejemplo de videojuego que utiliza algoritmos de *flocking* para mover sus unidades, http://goo.gl/zuRUdj, Útlima visita: 18/06/2015.
[34] Varios autores, Imágenes de *Flocking*, http://goo.gl/fPLHsp, Última visita: 18/06/2015
[35] Bakkes, S. C., Spronck, P. H., & van Lankveld, G. (2012). *Player behavioural modelling for video games. Entertainment Computing*, 3(3), 71-79.
[36] Palma, R., González-Calero, P. A., Gómez-Martín, M. A., & Gómez-Martín, P. P. (2011, March). *Extending Case-Based Planning with Behavior Trees.* In FLAIRS Conference.
[37] Ejemplo de videojuego que hizo famoso el uso de árboles de comportamiento, https://goo.gl/BvdqH5, Última visita: 18/06/2015.
[38] Ejemplo de videojuego que utiliza árboles de comportamiento, http://goo.gl/11xdVo, Última visita: 18/06/2015.
[39] Aung Sithu Kyaw, Cliffor Peters, Thet Naing Swe (2013), *Unity 4x Game AI Programming*, (*Behaviour Trees*).