Universitat de Girona
Escola Politècnica Superior

Treball final de màster

Estudi: Màster en Enginyeria Informàtica

Títol: Sistema de IA para la simulación de comportamientos de combate en videojuegos FPS

Document: **Memoria**

Alumne: **Mariano Agustín Trebino López**

Tutor: **Dr. Gustavo Patow**
Departament: **IMAE**
Àrea: **Llenguatges i sistemes informàtics**

Convocatòria (mes/any): 06/2016

~ *Agradecimientos* ~

Me gustaría agradecer especialmente a Gustavo por ayudarme en todo momentos con sus perspicaces comentarios y sobretodo por animarme en todo momento a no rendirme y a dar lo mejor de mi. Sin duda, has sido un gran apoyo y una de las principales razones por las cuales este proyecto ha acabado con éxito.

También me gustaría dar las gracias a mi pareja por los sacrificios hechos permitiéndome trabajar a todas horas, todos los días, fines de semana incluidos.

Por ultimo, querría agradecer a la comunidad de Unreal Engine por responder a la mayoría de mis preguntas relacionadas con los detalles de implementación de una manera totalmente desinteresada.

Índice

1 Introducción....9
1.1 Motivación....9
1.2 Objetivos....9
1.3 Planificación....10
1.4 Temporalización....11
2 Marco de trabajo....13
2.1 Introducción....13
2.2 IA en videojuegos....13
2.3 Toma de decisiones....15
2.3.1 Máquinas de estados finitas....15
2.3.2 Máquinas de estados finitas jerárquicas....16
2.3.3 Arboles de comportamiento....18
2.3.4 Sistemas de utilidad....22
2.3.5 Planificadores de acciones orientados a objetivos....23
2.3.6 Redes de tareas jerárquicas....25
2.4 Técnicas utilizadas en *FPS*....26
2.4.1 Introducción....26
2.4.2 *Tactical Pathfinding*....26
2.4.3 Predicción del jugador....27
2.4.4 Razonamiento espacial dinámico....28
2.4.5 Comportamientos colectivos....30
3 Entornos de trabajo analizados....36
3.1 Introducción....36
3.2 *Requisitos*....36
3.3 Opciones analizadas....37
3.3.1 *Source 2*....37
3.3.1.1 *Introducción*....37
3.3.1.2 Evaluación de requisitos....37
3.3.2 CryEngine....38
3.3.2.1 *Introducción*....38
3.3.2.2 Evaluación de requisitos....38
3.3.3 *Unity 3D (Unity 5)*....38
3.3.3.1 *Introducción*....38
3.3.3.2 Evaluación de requisitos....39
3.3.4 *Unreal Engine 4*....40
3.3.4.1 Introducción....40
3.3.4.2 Evaluación de requisitos....40
3.4 Opción escogida....41
4 Descripción del entorno de trabajo....44
4.1 *Introducción*....44
4.2 Unreal Engine 4....44
4.2.1 Sistema de percepción....44
4.2.2 Sistema de navegación....45
4.2.3 Behavior Trees....47

4.2.4 Environment Query System (EQS)....49
4.3 Juego base....50
4.3.1 Sistema de IA inicial....50
4.3.1.1 Sentir....50
4.3.1.2 Pensar....50
4.3.1.3 Actuar....50
5 Sistema de comportamiento....52
5.1 Introducción....52
5.2 Pensar....52
5.2.1 Árbol de comportamiento principal....52
5.2.1.1 Responsabilidad....52
5.2.1.2 Activación....52
5.2.1.3 Desactivación....52
5.2.1.4 Memoria....53
5.2.1.5 Funcionamiento....53
5.2.2 *Árbol de comportamiento Idle*....55
5.2.2.1 Responsabilidad....55
5.2.2.2 Activación....55
5.2.2.3 Desactivación....55
5.2.2.4 Memoria....55
5.2.2.5 Funcionamiento....55
5.2.3 Árbol de comportamiento Patrol....56
5.2.3.1 Responsabilidad....56
5.2.3.2 Activación....56
5.2.3.3 Desactivación....56
5.2.3.4 Memoria....56
5.2.3.5 Funcionamiento....56
5.2.4 *Árbol de comportamiento Search*....57
5.2.4.1 Responsabilidad....57
5.2.4.2 Activación....57
5.2.4.3 Desactivación....57
5.2.4.4 Memoria....57
5.2.4.5 Funcionamiento....58
5.2.5 *Árbol de comportamiento Fight*....59
5.2.5.1 Responsabilidad....59
5.2.5.2 Activación....59
5.2.5.3 Desactivación....59
5.2.5.4 Memoria....59
5.2.5.5 Funcionamiento....60
5.2.5.5.1 Hide Behavior....62
5.2.5.5.1.1.1 Responsabilidad....62
5.2.5.5.1.1.2 Activación....62
5.2.5.5.1.1.3 Desactivación....62
5.2.5.5.1.1.4 Funcionamiento....63
5.2.5.5.2 Push Behavior....63
5.2.5.5.2.1.1 Responsabilidad....63

5.2.5.5.2.1.2 Activación....63
5.2.5.5.2.1.3 Desactivación....64
5.2.5.5.2.1.4 Funcionamiento....64
5.2.5.5.3 Attack *Behavior*....65
5.2.5.5.3.1.1 Responsabilidad....65
5.2.5.5.3.1.2 Activación....65
5.2.5.5.3.1.3 Desactivación....66
5.2.5.5.3.1.4 Funcionamiento....66
5.3 Sentir....69
5.3.1 *Percepción*....69
5.3.1.1 Vista....69
5.3.1.2 Oído....69
5.3.2 Introspección....70
5.3.2.1 Estado....70
5.3.2.2 Munición....71
5.3.2.3 Vida....71
5.3.2.4 Combate....71
5.3.3 Cálculo....72
5.3.3.1 Visibilidad del jugador....72
5.3.3.2 Distancia al jugador....73
5.3.3.3 Localización del jugador....73
5.3.3.4 Jugador en paradero desconocido....73
5.3.3.5 Actualización de posiciones tácticas....73
5.4 Actuar....74
5.4.1 Servicios....74
5.4.1.1 Detección y actualización de datos....74
5.4.1.2 Disparar....74
5.4.1.3 Supresión....74
5.4.2 Tareas....75
5.4.2.1 Reload....75
5.4.2.2 Wait....75
5.4.2.3 Move To....75
5.4.2.3.1 Mejor camino: distancia....75
5.4.2.3.2 Mejor camino: seguridad....76
5.4.2.4 Cálculos tácticos....76
5.4.2.4.1 Punto de patrulla....77
5.4.2.4.2 Punto de búsqueda....77
5.4.2.4.3 Punto de cobertura....78
5.4.2.4.4 Punto de ataque....80
5.4.2.4.5 Punto de avance....81
5.4.2.4.6 Punto de supresión....82
5.4.2.4.7 Punto con linea de fuego....83
6 Implementación....86
6.1 Introducción....86
6.2 *Visibilidad*....86
6.2.1 Algoritmo 1....86

6.2.2 Algoritmo 2....87
6.3 Tactical pathfinding....94
6.3.1 Algoritmo 1....95
6.3.2 Algoritmo 2....96
6.3.2.1 División del segmento....96
6.3.2.2 Visibilidad de un punto....97
6.3.2.3 Cálculo de coste final....98
6.4 *Mapa de influencia*....100
6.4.1 Representación....100
6.4.2 Uso....101
6.4.3 Configuración....101
6.4.3.1 Momentum....101
6.4.3.2 Decay....101
6.4.3.3 Frecuencia de actualización....101
6.4.4 Implementación....102
6.4.4.1 Función EsNavegable....102
6.4.4.2 Función EsVisiblePorAlgunNPC....103
6.4.4.3 Función ObtenerConexionesNavegables....103
6.4.4.4 Función ActualizarEstructuraLocal....104
6.5 Disparos y supresión....104
6.5.1 Linea de Fuego....105
6.5.2 EQS: Punto con linea de fuego....106
6.5.3 EQS: Supresión....106
6.6 Tests EQS....106
6.6.1 Tests genéricos....106
6.6.1.1 Test de coste (pathfinding)....106
6.6.1.2 Test de distancia....107
6.6.1.3 Test de visibilidad....108
6.6.1.4 Test “detrás” de los obstáculos....109
6.6.1.5 Test de proximidad a los obstáculos....110
6.6.1.6 Test de aleatoriedad....111
6.6.2 *Tests* específicos....112
6.6.2.1 Test de influencia....112
6.6.2.2 Test de expansión....112
6.6.2.3 Test de flanqueo....113
6.6.2.4 Test de repetición....115
7 Contratiempos....117
7.1 Introducción....117
7.2 *Mala percepción*....117
7.3 Colisiones entre agentes....118
7.4 Configuración de los parámetros....121
7.5 Depuración....122
7.6 Situaciones de supresión....123
7.7 Problemas varios....124
8 Resultados....126
8.1 Introducción....126

8.2 Propagación de la influencia....126
8.3 Propagación de la influencia y visibilidad....127
8.4 Comportamiento Patrol....127
8.5 *Comportamiento Search*....128
8.6 *Comportamiento* Fight....129
8.6.1 Rama de cobertura....129
8.6.2 Rama de avance....130
8.6.2.1 *Supresión*....131
8.6.3 Rama de ataque....131
8.6.3.1 Ataque en grupo....132
8.7 *Gameplay*....133
8.7.1 Un jugador contra un agente....133
8.7.2 *Un jugador contra cuatro agentes*....134
9 Conclusiones....137
9.1 Introducción....137
9.2 Evaluación del resultado....137
9.2.1 Cumplimiento de los objetivos....137
9.2.1.1 Tactical pathfinding....137
9.2.1.2 Razonamiento espacial dinámico....138
9.2.1.3 *Predicción del jugador*....138
9.2.1.4 Coordinación colectiva....138
9.2.1.5 Rendimiento....139
9.2.2 Descripción del resultado....139
9.3 Desarrollo....140
9.3.1 Inteligencia artificial y videojuegos....140
9.3.2 Unreal Engine 4....141
9.3.3 Experiencia personal....141
9.4 *Punto final*....142
10 Trabajo futuro....144
10.1 Comportamiento grupal....144
10.2 Supresión....144
10.3 Mapa de influencia....144
10.4 Diversidad....145
10.5 Fuego amigo....145
10.6 Malla de navegación dinámica....145
10.7 Mirar alrededor....146
10.8 Predicción del jugador....147
10.9 Rendimiento....147

01

Introducción

1 Introducción

1.1 Motivación

La contínua mejora de la tecnología y la elaboración de dispositivos más potentes año tras año permite el desarrollo de videojuegos con mejores gráficos, mejor jugabilidad y, en general, entornos virtuales que se asemejan cada vez más a la realidad. No es de extrañar entonces que la industria de los videojuegos haya crecido significativamente en la última década. De hecho, en el año 2013, los beneficios generados por los videojuegos duplicaron el generado por la industria del cine.

La simulación de estos entornos virtuales cada vez más realistas permite a los jugadores una experiencia de juego mucho más rica que la que podían experimentar hace unos años. La mejora de esta experiencia viene dada por una estrecha y perfecta colaboración de diferentes componentes. Uno de los más importantes es la **inteligencia artificial**, aunque también hay muchos otros, como el comportamiento físico de los elementos, el sonido, la historia, interactividad, etc.

En mi caso, la inteligencia artificial de los videojuegos FPS a los que he jugado nunca ha cumplido mis expectativas y siempre la he percibido muy pobre y predecible. Nunca he llegado a sumergirme completamente en esta experiencia virtual.

Por estos motivos, y dado a que siento un especial interés por los videojuegos, especialmente por los *FPS* (*First Person Shooter*) así como por la inteligencia artificial, he decidido desarrollar un proyecto que consistiese en crear la IA de un *FPS* que se comporte de manera realista.

1.2 Objetivos

El objetivo de este proyecto es enriquecer la inteligencia artificial de un juego FPS existente para que simule el comportamiento humano en situaciones de combate. Dado que “simular el comportamiento humano” es algo amplio, he descompuesto este objetivo principal en cinco objetivos más pequeños fácilmente alcanzables. Bajo mi punto de vista, cumplir estos objetivos es lo que diferencia a un jugador mediocre de un buen jugador, y en este caso, a un agente “*dummy*” (tonto) de un agente inteligente.

El primer objetivo, totalmente imprescindible en todo videojuego FPS, es el ***Tactical Pathfinding.*** En situaciones de combate, la inteligencia artificial tiene que ser capaz de moverse de cobertura en cobertura de manera rápida y efectiva al igual que lo haría un jugador real. Por lo tanto, a la hora de evaluar los caminos que llevan a un punto determinado, lo ha de hacer desde un punto de vista táctico. Es decir, no sólo

ha de evaluar la distancia euclidiana como lo haría en una situación normal, sino que también ha de tener en cuenta las amenazas y lineas de fuego enemigas/amigas, entre otros.

El segundo objetivo es el **razonamiento espacial dinámico***.* La inteligencia artificial debe ser capaz de entender el entorno, captando zonas de cobertura o zonas de ataque así como detectar de manera rápida los cambios en el ambiente y responder de manera inmediata con acciones eficaces.

El tercer objetivo es la **predicción del jugador**. Para nosotros, resulta intuitivo y muy fácil saber en qué posición esta un jugador dada su posición inicial y un tiempo determinado. Para los agentes inteligentes no lo es. Por lo tanto, es muy importante dotarlos de una herramienta que les permita predecir la localización del jugador cuando éste no es visible.

Por otro lado, es vital conseguir una **coordinación colectiva** entre los agentes proporcionándoles objetivos comunes, que deben alcanzar a partir de la cooperación y el uso de sus diferentes habilidades. La mayor dificultad radica en ser capaz de detectar cuándo y cómo activar la coordinación colectiva sobre el comportamiento individual.

Por último, es importante preservar en todo momento un **nivel alto de rendimiento** para que la experiencia de juego no sea vea afectada negativamente por un mal diseño o una pobre implementación de los algoritmos relacionados con el sistema de inteligencia artificial.

Alcanzar todos y cada uno de estos objetivos nos permitirá crear una ilusión de inteligencia que contribuirá notablemente en una mejora de la experiencia de juego, suspendiendo al jugador en este entorno virtual.

1.3 Planificación

Con el fin de cumplir los objetivos propuestos en el apartado anterior, he definido un plan de trabajo muy simple pero efectivo. Este plan está formado por cinco grandes fases:

Fase 1: Entorno de trabajo y juego base. Evaluar cual es la mejor opción entre los diferentes motores gráficos que existen actualmente así como la existencia de un juego base de tipo *FPS* que los acompañe que sea fácilmente accesible y me permita modificar su sistema de IA.

Fase 2: Estudio del entorno y del juego base. Estudiar el entorno escogido así como el código fuente del juego base, su sistema de IA y las posibilidades que ofrece.

Fase 3: Comportamiento inteligente individual. Diseñar e implementar el comportamiento individual que comprende todo aquello que no requiera la

interacción con otro agente inteligente.

Fase 4: Comportamiento inteligente colectivo. Diseñar e implementar las diferentes fases de la inteligencia colectiva así como las modificaciones pertinentes en la inteligencia individual. Esta inteligencia tiene que permitir que los agentes se coordinen y cooperen para llevar a cabo un comportamiento en grupo.

Fase 5: Documentación. Elaboración de la memoria detallando todo el trabajo realizado.

1.4 Temporalización

Por último, la planificación previa se materializa en la Tabla 1: La temporalización inicial de las diferentes fases del proyecto donde se detalla el inicia, final y duración estimada para cada fase.

	Febrero	Marzo	Abril	Mayo	Junio
Fase 1	X				
Fase 2	X				
Fase 3	X	X	X		
Fase 4			X	X	
Fase 5				X	X

Tabla 1: La temporalización inicial de las diferentes fases del proyecto

02

Marco de trabajo

2 Marco de trabajo

2.1 Introducción

La inteligencia artificial en los videojuegos es tan antigua como los videojuegos mismos. Recordemos que años atrás los juegos multijugador no eran tan comunes y que por tanto, había una mayor cantidad de videojuegos con un sistema de IA. No obstante, hoy en día, todos los juegos siguen conservando un componente fundamental de IA.

Con transcurso de los años el diseño y la implementación de sistemas de inteligencia artificial para videojuegos se ha vuelto más y más complicado a medida que las expectativas de los usuarios aumentan.

Como consecuencia, el rango de comportamientos que los *NPC (Non-Player Characters)* pueden mostrar es muy amplio. Desde simples animales a los que dar caza a compañeros de viaje que te acompañan durante horas de interacción. Aunque todos ellos siguen, a alto nivel, un comportamiento similar, el proceso de toma de decisiones aún está poco definido y en muchas ocasiones puede resultar ambiguo. Dependiendo del género del juego, del dispositivo o de la audiencia, entre otros, un sistema de IA perfectamente válido para un juego puede no serlo para otro.

A lo largo de este capitulo definiremos concretamente qué es la IA en un videojuego, así como los sistemas y técnicas más populares para modelar comportamiento. También estudiaremos los problemas y las soluciones más comunes de la IA en los juegos de tipo *FPS.*

2.2 IA en videojuegos

Wikipedia define la inteligencia artificial como el estudio y diseño de agentes inteligentes, donde un agente es un sistema que percibe el entorno y toma acciones para maximizar sus probabilidades de éxito [1]. Clásicamente la arquitectura de estos agentes inteligentes se ha divido en tres fases inter-relacionadas [2] (ver):

- **Sentir.** Hace referencia a cómo el agente artificial siente u obtiene datos del entorno (normalmente mediante sensores) y los transforma en información, como por ejemplo, munición, posición actual, cobertura más cercana, etc.
- **Pensar.** Consiste en el proceso de toma de decisiones y posterior actuación a partir del análisis de los datos obtenidos durante la fase de Sentir.
- **Actuar.** Se refiere a la ejecución de algún tipo de acción que repercute y modifica el agente o el entorno de alguna manera, como por ejemplo moverse, disparar, saltar, etc.

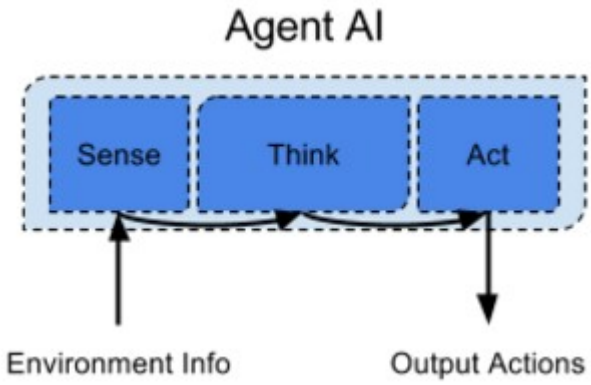


Figura 2.1: Esquema describiendo el proceso de Pensar-Sentir-Actuar de un agente inteligente

Aunque esta no es, ni mucho menos, la única definición de la inteligencia artificial, la describe de manera muy acertada.

Concretamente, el objetivo de la IA en el ámbito de los videojuegos es el de crear una **ilusión de inteligencia** para el jugador, en lugar de centrarnos en crear una verdadera inteligencia, tal y como existe en los seres humanos [3]. Cada juego es diferente y las necesidades de la inteligencia artificial y de los diferentes componentes del juego pueden variar ampliamente. No obstante, siempre hay un objetivo común: **apoyar la experiencia de juego** utilizando las técnicas más apropiadas, sean las que sean.

Los jugadores son participantes voluntarios que desean involucrase en la realidad que estamos creando para ellos. Quieren sumergirse en nuestra ilusión y creer que todo es real. Por lo tanto, es nuestra la responsabilidad de crear un entorno que los mantenga en esta ilusión y **suspensión de incredulidad**. Tenemos éxito en el momento en el que el jugador piensa o percibe la IA como si fuera real, aunque el algoritmo sea muy simple. No obstante, podemos fracasar en el mismo momento en el que la IA recuerda al usuario que es solo un programa informático y no es real. Por este motivo, es fundamental evitar comportamientos erróneos y la llamada "**estupidez artificial"**, es decir, seleccionar una acción incorrecta o que simplemente no tiene ningún sentido, como por ejemplo, caminar contra las paredes, quedarse atascado en la geometría o ignorar a un jugador que te esta disparando.

La solución obvia para evitar esta estupidez artificial es simplemente mejorar la IA. No obstante, a veces esto puede ser muy complicado ya que las expectativas de los diferentes jugadores pueden ser muy diversas que resulta difícil cumplirlas todas.

Por esta razón, en la inteligencia artificial al igual que en el desarrollo de software, **no hay una solución única y universal**. Algunos juegos de estrategia como *The*

Sims requieren sistemas más reactivos mientras que otros prefieren sistemas más predecibles, como *World of Warcraft*. Ninguno de ellos es malo, simplemente son juegos diferentes que proporcionan experiencias distintas. Es importante tener en cuenta el tipo de juego y la experiencia que queremos proporcionar a la hora de escoger el sistema de toma de decisiones de nuestra inteligencia artificial.

2.3 Toma de decisiones

El hecho de que no exista un único sistema universal e infalible ha provocado que emerjan diversos algoritmos diferentes para seleccionar el comportamiento de la IA Es necesario entonces que los diseñadores y programadores conozcan cada herramienta con el fin de utilizarla cuando mejor convenga. Qué algoritmo es mejor para un *NPC* dependiendo del genero del juego, la plataforma, la audiencia, el entorno y otros muchos aspectos.

A continuación explicaré de manera general las diferentes herramientas utilizadas en los últimos años por los programadores de IA para modelar el comportamiento de los *NPC.*

2.3.1 Máquinas de estados finitas

Las máquinas de estados finitas o *FSM* son muy conocidas en el mundo de la informática y ampliamente utilizadas para modelar comportamientos en la inteligencia artificial [3] [4]. Conceptualmente son simples y rápidas de programar, ademas de muy potentes y flexibles.

Las *FSM* dividen el comportamiento de un *NPC* en piezas más pequeñas llamadas **estados**. Cada estado representa un comportamiento especifico o una configuración de información determinada y solo puede haber un estado activo a la vez. Los estados están conectados mediante **transiciones dirigidas** que permiten pasar de un estado a otro si un conjunto de condiciones se satisface.

Una de las características más atractivas de las *FSM* es que son muy fáciles de entender a partir de un simple esquema (ver Figura 2.2).

En este caso, el guardia comenzaría en el estado inicial *Patrol* durante el cual seguiría una ruta pre-definida pero prestando atención al entorno. Si escucha algún sonido pasaría al estado de *Investigate* durante un tiempo antes de volver al estado *Patrol.* Si en algún momento visualiza a un enemigo, pasaría al estado de *Attack* para hacer frente a la amenaza. Si durante el ataque pierde mucha vida pasaría al estado *Flee.* Si por el contrario, vence al enemigo, volvería al estado inicial de *Patrol.*

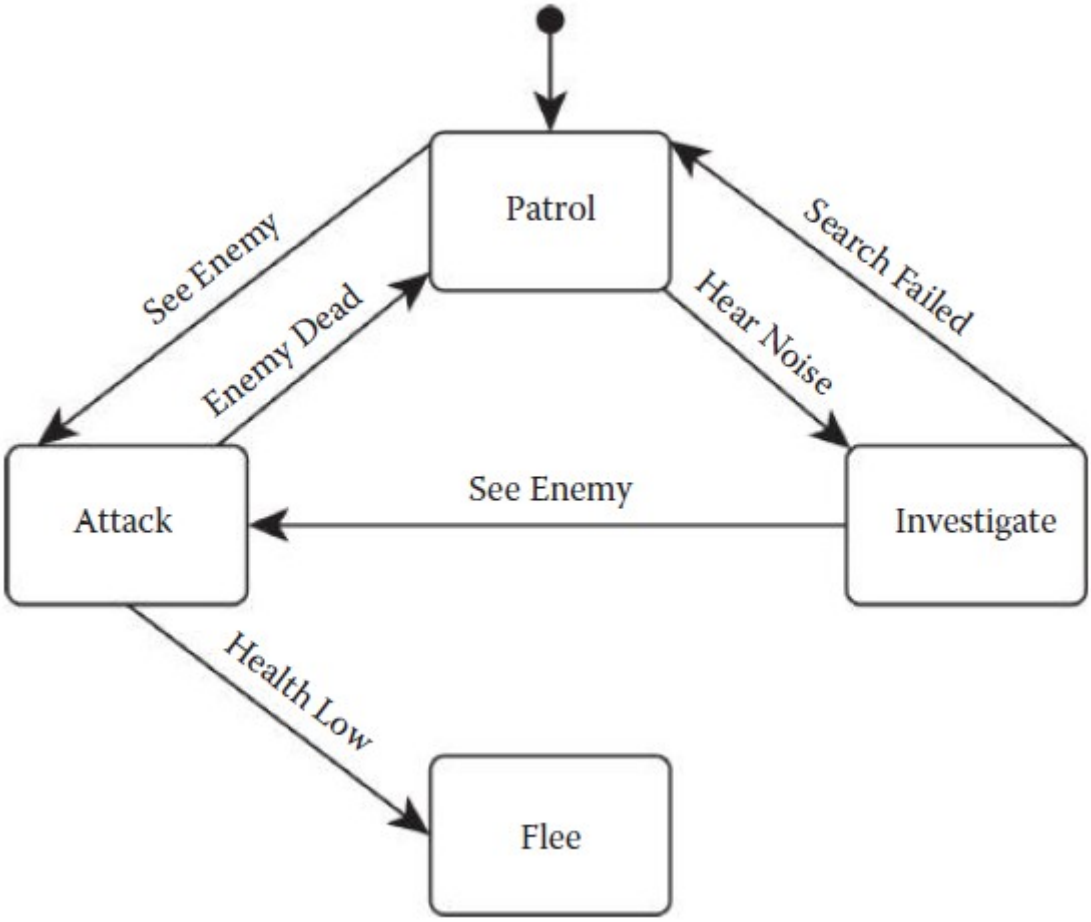


Figura 2.2: Esquema FSM representando el comportamiento de un guardia.

Actualmente existen diversas implementaciones diferentes de las *FSM.* No obstante, en todos los casos, debemos definir en cada estado, ademas de definir las transiciones posibles, tres funciones:

- *OnEnter*(). Que hacer cuando se entra en el estado.
- *OnUpdate*(). Que hacer mientras el estado esta activo y no se realiza ninguna transición.
- *OnExit*(). Que hacer cuando se sale del estado.

Por lo que respecta a los demás detalles de implementación, considero que no son relevantes y que el funcionamiento de las *FSM* queda suficientemente ilustrado con el ejemplo anterior.

2.3.2 Máquinas de estados finitas jerárquicas

Como hemos visto en el apartado anterior las *FSM* son muy útiles para modelar la IA de un *NPC,* aunque tienen una importante debilidad. Cuando la maquina de estados

crece hasta alcanzar 20 o 30 estados, el numero de transiciones crece exponencialmente y se vuelve muy compleja y propensa a fallar ante cualquier modificación [3] [5] .

Ademas, las *FSM* tampoco permiten gestionar el re-uso de comportamientos, provocando duplicidad de estados, de código, e incrementando la complejidad del sistema. Por ejemplo, supongamos que tenemos un *NPC* con un comportamiento muy básico que simplemente consiste en patrullar de la puerta a una zona segura (ver Figura 2.3).

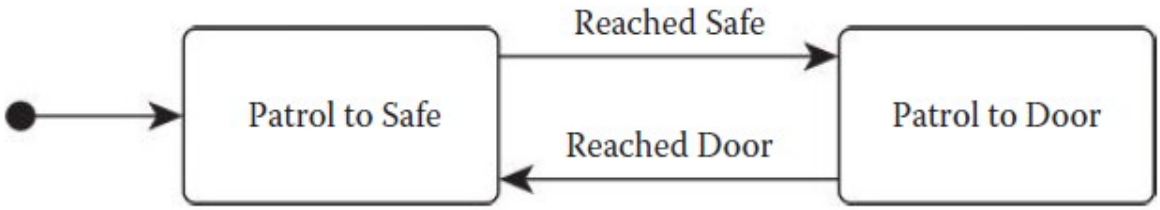


Figura 2.3: Esquema FSM representando un comportamiento simple de patrulla

Deseamos agregar la posibilidad de que en cualquier momento, el *NPC* pueda responder a una llamada, mantener una conversación y volver a su estado de patrulla original. Es decir, si el *NPC* se encuentra en el estado *Patrol to Door,* después de la llamada debe volver al estado *Patrol to Door.* De la misma manera, si esta en el estado *Patrol to Safe* al finalizar la llamada debe volver al estado *Patrol To Safe*.

Para solucionar este problema utilizando una *FSM* debemos agregar dos estados *Conversation.* En una *FSM* algo más compleja podrían ser incluso más. Aqui se hace presente la dificultar que presentan las *FSM* al re-utilizar comportamientos similares (en este caso incluso iguales) y el número de estados puede crecer de manera muy rápida (ver Figura 2.4).

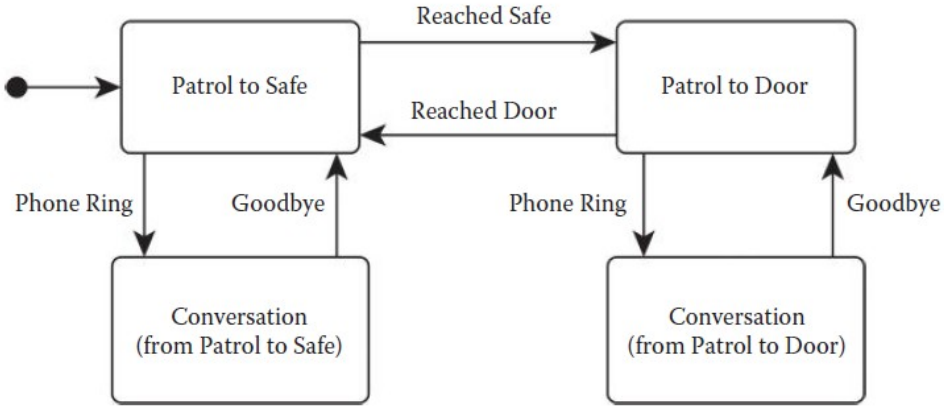


Figura 2.4: Esquema FSM con múltiples instancias del estado Conversation

Afortunadamente, las máquinas de estado jerárquicas o *HFSM* solucionan estos problemas. En una *HFSM* cada estado puede representar una máquina de estados completa. Esto permite dividir una máquina de estados en múltiples máquinas dispuestas de manera jerárquica, facilitando la re-utilización de código y la escalabilidad.

Volviendo al ejemplo anterior, vemos como se puede solucionar evitando la duplicidad si anidamos nuestros dos estados de patrulla en un estado *Watch Building* y después simplemente lo unimos el estado de *Conversation (*ver Figura 2.5*).*

Los algoritmos utilizados por las *HFSM* son muy similares a los que usan las *FSM* con la dificultad añadida de una actualización recursiva provocada por la anidación de estados de manera jerárquica.

Tanto las *FSM* como las *HFSM* son increíblemente potentes y útiles para desarrollar diferentes tipos de comportamientos. No obstante, en algunos casos en los que el numero de transiciones es muy elevado y cada estado se une con prácticamente todos los otros, ni una *FSM* o *HFSM* nos ayudara. En estos casos tendremos que buscar otras técnicas como las que comentamos a continuación.

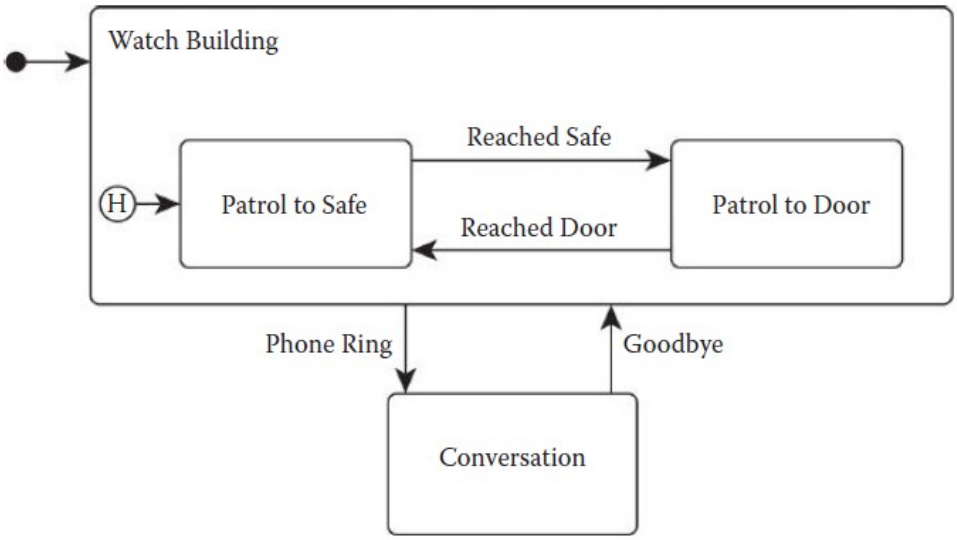


Figura 2.5: Esquema de una HFSM solucionando la duplicidad de estados

2.3.3 Arboles de comportamiento

Los arboles de comportamiento o *Behavior Trees (BT)* surgen con la motivación de solucionar algunos de los problemas pendientes de las *FSM* y las *HFSM.* Aunque las *HFSM* resuelven algunos de los problemas de las *FSM*, como la escalabilidad y la re-utilización de estados, presentan algunos otros [3] [6] [7]. Por ejemplo, re-utilizar estados y transiciones no es ni mucho menos trivial, si no que requiere muchísima

experiencia. Ademas, tampoco permiten ningún tipo de modularidad para los estados: no permiten re-utilizar estados con el fin de alcanzar diferentes objetivos sin tener que crear más y más transiciones.

Los *BT* representan una aproximación diferente con el objetivo de incrementar la modularidad de los estados encapsulando la lógica de manera transparente para que puedan ser re-utilizados en diferentes partes, manteniendo la escalabilidad y simplicidad de las *FSM* y *HFSM.*

Al igual que en las técnicas anteriores, el funcionamiento de los árboles de comportamiento es muy simple. La estructura comienza en un nodo raíz o *root* el cual esta formado por comportamientos y cada comportamiento a su vez esta formado por el comportamiento de sus hijos, lo que crea esta estructura en forma de árbol. De hecho, si lo pensamos atentamente, un *BT* no es más que una maquina de estados en la cual se eliminan las transiciones a los estados externos de manera que los estados se convierten en auto-contenidos. De hecho, una vez conseguido esto, los estados dejan de ser estados porque desaparecen las transiciones y pasan a ser comportamientos (ver Figura 2.6). De esta manera, añadir o remover un nuevo comportamiento no afecta al resto del árbol, contrariamente a lo que sucedía en las *FSM* donde cada estado debía conocer el criterio de transición a cada uno de los otros estados.

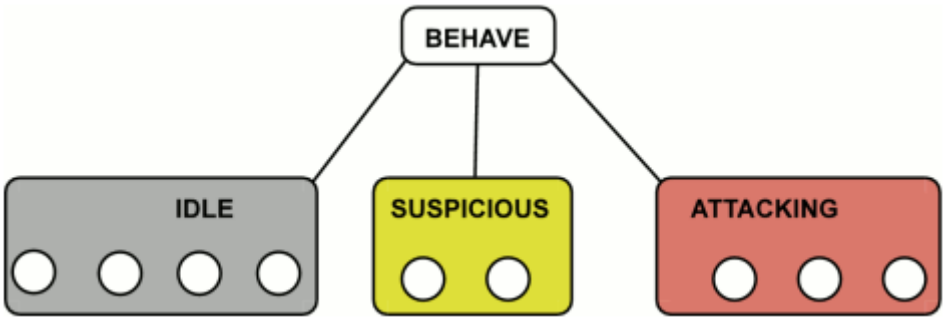


Figura 2.6: BT formado por diversos comportamientos modulares

Pero ¿Qué es exactamente un comportamiento? Un comportamiento esta definido por una **precondición**, la cual especifica cuando se ejecuta el comportamiento y una **acción o acciones**, especificando exactamente que hacer cuando el comportamiento esta activo.

El algoritmo comienza en el nodo raíz y evalúa las precondiciones de los diferentes comportamientos de izquierda a derecha (de más a menos prioritario), decidiendo cual se tiene que ejecutar. En cada nivel solo puede haber un comportamiento ejecutándose, de manera que si un comportamiento se ejecuta, ninguno de sus hermanos se revisaran. Contrariamente, si la precondición no se cumple, automáticamente el algoritmo omite a todos los hijos de este comportamiento y pasa

al siguiente hermano. Una vez se ha evaluado todo el árbol, los nodos hoja evaluados satisfactoriamente son los que se tienen que ejecutar.

Por lo tanto, los **nodos hoja** son las tareas o acciones concretas que el agente puede ejecutar y que modifican el entorno o el estado del agente de alguna manera.

Los nodos con múltiples hijos se llaman **nodos compuestos** debido a que siguen el patrón de diseño de software *composite,* el cual especifica como un conjunto de objetos se debe juntar en colecciones para construir complejidad. En este caso, lo que hacemos es crear comportamientos más inteligentes a partir de combinar comportamientos simples. Los nodos compuestos más comunes son:

Los nodos compuestos de tipo **secuencia** ejecutan sus hijos en un orden determinado, que implícitamente demuestra las dependencias entre los comportamientos (ver Figura 2.7) [8]. Este nodo solo ejecuta el siguiente hijo si el anterior ha tenido éxito, y la secuencia unicamente tiene éxito si todos sus hijos se han ejecutado con éxito.

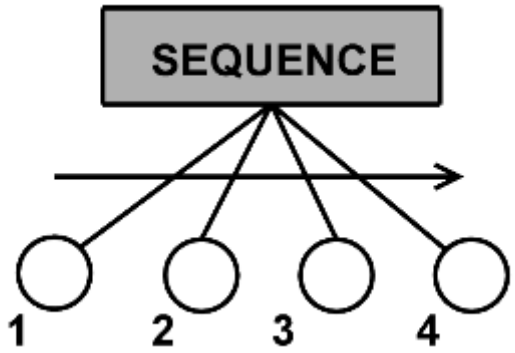


Figura 2.7: Nodo compuesto de tipo secuencia formada por cuatro comportamientos

Los nodos compuestos de tipo **selector** evalúan las condiciones de ejecución de sus nodos hijos de izquierda a derecha y ejecutan únicamente el primero que las cumpla (ver Figura 2.8) [9]. El selector tiene éxito si ejecuta un nodo hijo con éxito.

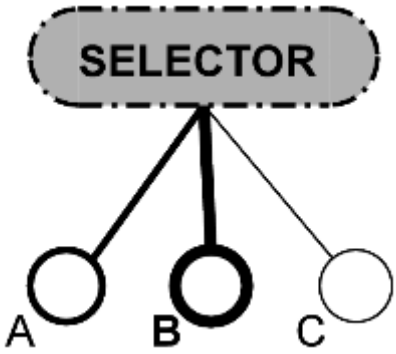


Figura 2.8: Nodo compuesto de tipo selector con tres hijos que está ejecutando el segundo hijo

El último tipo de nodo compuesto es el nodo **paralelo**. Como su nombre indica, este nodo permite ejecutar dos comportamientos de manera simultanea (ver Figura 2.9). Dada su naturaleza, estos nodos suelen ser más complejos que los dos anteriores y su implementación puede variar dependiendo del entorno. Normalmente es necesario configurar la política de fallada, es decir, cuándo los hijos deben fallar para que el nodo acabe fallando; y la de finalización, cuántos nodos deben completarse con éxito para finalizar el nodo compuesto con éxito. Lo más común es que en ambos casos se implementen dos opciones: todos o sólo uno.

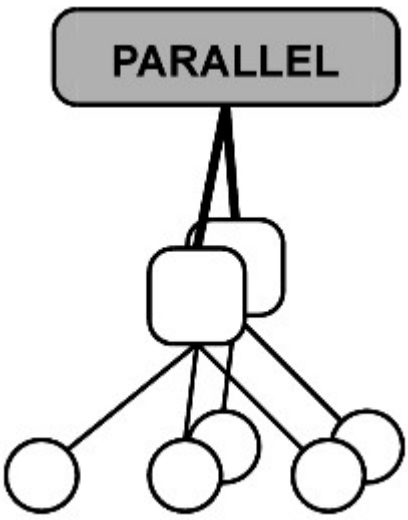


Figura 2.9: Nodo compuesto de tipo paralelo ejecutando dos comportamientos simultáneamente

Por otro lado, es muy importante destacar la existencia y el uso de **decoradores** para añadir funcionalidad a cualquier tipo de comportamiento de manera totalmente independiente de su funcionalidad (sin saber exactamente qué hace) [10]. Esta característica está inspirada en el patrón de diseño de software decorador.

Los decoradores funcionan sobre cualquier comportamiento y hay de diferentes tipos:

- Filtros. Limitan el numero de veces que un comportamiento se ejecuta, añaden un temporizador para evitar ejecuciones muy frecuentes, etc.
- Controladores. Controlan los errores y reinician las ejecuciones, acceden a información,etc.
- Modificadores de flujo. Modificar el resultado de una operación o fuerzan un determinando comportamiento para que se ejecute.
- Meta operadores. Operaciones de depuración, log, visualización, etc.

Dicho esto, resulta muy fácil entender la popularidad de los *BT. S*u elevada simplicidad, facilidad de implementación y potencia, así como extensibilidad, los convierte en herramientas muy valiosas para el desarrollo de IA.

Al final, los *BT* son simplemente otra manera de organizar una *FSM.* Cualquier *BT* se puede convertir a una *FSM* si fuese necesario. La diferencia consiste en que mientras que una *FSM* proporciona un mayor control sobre las transiciones y estados, el *BT* permite trabajar con una lógica más modular, simple y fácil de reutilizar, aunque suelen ser más lentos. La diferencia en el tiempo de ejecución viene dada a que el *BT* tiene que evaluar muchas más condiciones y comportamientos que una *FSM.*

2.3.4 Sistemas de utilidad

Mucha parte de la lógica de la IA, y de la computación en general, esta basada en respuestas booleanas, como por ejemplo: ¿Puedo ver al enemigo?, ¿Estoy sin munición?, etc. Normalmente, estas preguntas o conjunto de preguntas suelen mapease directamente a una única acción.

- ¿Puedo ver al enemigo? → Atacar
- ¿Estoy sin munición? → Recargar
- ¿Estoy sin munición? Y ¿Puedo ver al enemigo? → Esconder

No obstante, en muchas ocasiones podemos encontrarnos con que una respuesta booleana no es apropiada. Pensemos en cuan lejos está un enemigo, cuántas balas le quedan o el porcentaje de vida actual. En estos casos en los que se trabaja con valores contínuos, nos interesa saber que tanto deseamos ejecutar una acción en vez de simplemente ejecutarla o no.

Los sistemas de utilidad o *Utility Systems (US)* **miden cuánto preferimos una acción** de un conjunto de posibles acciones [3]. Aunque normalmente los sistemas de utilidad se utilizan como complemento de otros sistemas, como por ejemplo, en decoradores de *BT*, también es posible utilizarlos, incluso a veces preferible, para construir sistemas de IA completos.

Comúnmente estos sistemas se utilizan en juegos en los que la IA puede realizar diversas posibles acciones y no hay una que sea perfecta. En estos casos, utilizamos la **utilidad** para guiar al proceso de toma de decisiones, midiendo y decidiendo qué acciones son preferibles en cada situación.

Un ejemplo muy claro de estos sistemas se puede ver en el juego *The Sims.* En este juego, los personajes puntúan cada acción a partir de la información de su entorno (posición de la nevera, calidad de la cama, etc) y su información interna (nivel de hambre, higiene, aburrimiento, etc) para encontrar la que mejora su estado global. Supongamos que el *Sim* tiene mucho hambre pero que la comida esta quemada. En este caso, la acción es mucho más atractiva que si tuviese poco hambre. De la misma manera, si la comida fuese de una calidad suprema, seria mucho más atractiva incluso aunque tuviese poco hambre.

La principal ventaja de estos sistemas es que permiten una alta adaptabilidad dado que la manera de ponderar o puntuar las acciones se puede cambiar de manera dinámica. Es decir, a medida que el juego progresa y evoluciona, la IA también lo hace razonando de forma diferente y por lo tanto, tomando acciones distintas. Esto permite generar comportamientos más dinámicos y emergentes resultando en mejores experiencias de juego.

No obstante, como ya hemos visto anteriormente, estos sistemas de carácter más dinámico son por naturaleza impredecibles y tienen el inconveniente de ser más difíciles de controlar y supervisar. Por otro lado, ponderar con pesos las diferentes acciones es muchas veces más un arte que una ciencia, por lo que suele ser una tarea más empírica (prueba y error) para que el mejor comportamiento brille en cada momento sobre el resto.

Como ya hemos comentado en diversas ocasiones, cada sistema tiene sus ventajas y sus inconvenientes y nuestra tarea es la de elegir el sistema que mejor se adapte a nuestro juego.

2.3.5 Planificadores de acciones orientados a objetivos

Abandonando los modelos basados en estados, comportamientos o en utilidad surgieron los *Goal-Oriented Action Planning (GOAP)* por primera vez en el juego *F.E.A.R* en el año 2005 y que ha ido ganando adeptos como los títulos *Just Cause 2, Deus Ex: Human Revolution,* etc [3] [11].

GOAP es un concepto que deriva del *Standford Research Institute Problem Solver*

(*STRIPS),* un sistema de planificación desarrollado a principios de los 70. La idea principal, tanto *STRIPS* como de los sistemas *GOAP* es proveer al sistema de un conjunto de acciones describiendo el funcionamiento del juego. Estas acciones están formadas por unas **precondiciones**, que tienen que ser validas para ejecutarse y unos **efectos** que modifican el entorno.

Este sistema de planificación consta de un estado inicial, formado por un conjunto de hechos (información) en el que se encuentra el mundo y un estado final objetivo al que los *NPCs* quieren llegar (también formado por un conjunto de hechos). El sistema entonces tiene que determinar un plan (una secuencia de acciones) que permitirán a los *NPCs* cambiar del entorno del estado inicial al estado objetivo.

Aunque no consideramos relevante comentar los detalles de implementación del algoritmo, si considero importante detallar que utiliza la técnica ***backwards chaining search*** en contraposición con la técnica que todos conocemos, *forward chaining search* la cual se basa en heurísticas, podas, etc. La *backwards chaining search* consiste en comenzar desde el estado objetivo, buscando que acciones son necesarias para conseguirlo y las acciones necesarias para satisfacer las precondiciones de estas acciones. Este proceso se repite (del final hacia el principio) hasta llegar al estado inicial (ver Figura 2.10).

- Add the goal to the outstanding facts list
- For each outstanding fact
  - Remove this outstanding fact
  - Find the actions that have the fact as an effect
  - If the precondition of the action is satisfied,
    - Add the action to the plan,
    - Work backwards to add the now-supported action chain to the plan
  - Otherwise,
    - Add the preconditions of the action as outstanding facts

Figura 2.10: Pseudocódigo simplificado de un planificador GOAP

Una de las características fundamentales que diferencia los *GOAP* de los sistemas anteriores es que los planes especifican que hacer pero no necesariamente como hacerlo. A partir del entorno y un conjunto de acciones, es el agente quien piensa por sí mismo qué acciones son las más adecuadas para conseguir su objetivo. Esto permite que comportamientos dinámicos e interesantes emerjan de manera natural. No obstante, el nivel del control y supervisión que los diseñadores y programadores tienen sobre la IA, es mucho más bajo. En algunos casos, la satisfacción de los planes individuales de los agentes pueden ir en contra con el objetivo máximo de crear una experiencia de juego inmersiva y atractiva.

2.3.6 Redes de tareas jerárquicas

Aunque *GOAP* fue de los primeros (si no el primero) planificadores usados para modelar la IA en un videojuego, en los últimos años otros han ganado popularidad. Uno de estos sistemas son las redes jerárquicas de tareas o *Hierarchical Task Networks* (*HTN) [3] [12] [13].* Las *HTN* introducen una mejora gracias a su distribución jerárquica de las redes de tareas simples, *Simple Task Networks (STN)* así como lo hacían las *HFSM* con las *FSM.*

Las *HTN* han sido utilizadas por títulos tan conocidos como *KillZone 2, Transformers: Fall of Cybertron* y *Warcraft II [14]*. La única diferencia entre *HTN* y otros planificadores es la manera de encontrar el plan deseado.

Para ello, los *HTN* comienzan con el estado inicial del mundo y una tarea representando el problema que intentamos resolver. Esta tarea de alto nivel es descompuesta en tareas más y más pequeñas hasta que conseguimos un plan de tareas que es directamente ejecutables y que resuelve nuestro problema. Cada tarea de alto nivel puede ser llevada a cabo de diferentes maneras (de aquí la jerarquía) por lo que el estado inicial es el que determina el conjunto de tareas en el que se dividirán las tareas de alto nivel.

De manera contraria a como funcionaba *GOAP*, vemos que los *HTN son forward planners* ya que parten de un estado inicial y buscan el plan que nos lleva al estado final.

Como ya avanzamos antes de alguna manera, las tareas de los *HTN* pueden ser de dos tipos:

- **Tareas primitivas.** Son la unidad mínima en un *HTN* ya que son directamente ejecutables por los agentes y se utilizan para resolver el problema ya que afectan de manera directa al entorno. Por ejemplo: Disparar, Recargar, Moverse a una Cobertura, etc.
- **Tareas compuestas.** Son las tareas a las que antes nos referíamos como tareas de alto nivel ya que se pueden llevar a cabo de diversos modos, llamados **métodos**. Un método esta formado por un conjunto de tareas que cumplen una tarea compuesta así como por sus precondiciones.

Como resultado, tenemos un sistema que a partir del estado del mundo y un conjunto elevado de tareas compuestas dispuestas de manera jerárquica, es capaz de construir un plan de tareas primitivas que solucione el problema (ver Figura 2.11).

- Add the root compound task to our decomposing list
- For each task in our decomposing list
  - Remove task
  - If task is compound
    - Find method in compound task that is satisfied by the current world state
    - If a method is found, add method's tasks to the decomposing list
    - If not, restore planner to the state before the last decomposed task
  - If task is primitive
    - Apply task's effects to the current world state
    - Add task to the final plan list

Figura 2.11: Pseudocódigo simplificado de un planificador que utiliza HTN

Para ilustrar el funcionamiento utilicemos un ejemplo. Imaginemos que tenemos un un *NPC* y queremos que se comporte como un soldado: la tarea raíz se llamara *SerSoldado.* Dado que nos interesa que el *NPC* se comporte de manera diferente si hay un enemigo o no, definiremos dos métodos que describan estas situaciones. En el primer caso tendríamos un método *AtacarEnemigo* que, por ejemplo, podría disparar al enemigo si tiene un rifle con munición. Si no tuviese munición podría correr hacia él y atacarlo con un cuchillo.

Como vemos, cuanto más profundizamos en el comportamiento del soldado, más jerarquías creamos y refinamos. Finalmente se acaba creando una estructura que describe de manera muy intuitiva el comportamiento del agente.

Una de las principales ventajas, al igual que pasaba con *GOAP,* es que los personajes razonan por sí mismos de manera natural, en este caso, a través de la estructura jerárquica para encontrar el plan que soluciona el problema. No obstante, la elevada expresividad con la que se describe del comportamiento impide a los *NPC* crear planes que el diseñador no hubiese pensado previamente.

Por otro lado, el uso de tareas primitivas permite crear un diseño modular muy fácilmente re-utilizable en diferentes personajes.

2.4 Técnicas utilizadas en *FPS*

2.4.1 Introducción

Previamente hemos visto de manera rápida las técnicas más utilizadas para modelar los comportamientos de la IA en los diferentes videojuegos. A continuación, revisaremos los problemas más comunes de los sistemas inteligentes en los juegos de tipo *FPS* así como los procedimientos más utilizados para solucionarlos.

2.4.2 *Tactical Pathfinding*

Previamente, en la declaración de objetivos (ver apartado 1.2), ya señalábamos a

uno de los problemas más comunes a los que se enfrenta la IA en los juegos de tipo *FPS:* Es fundamental que los personajes sepan capaz de moverse de manera táctica en situaciones de combate, evitando las lineas de fuego enemigas y otras amenazas.

La mayoría de algoritmos de *pathfinding* utilizan implementaciones basadas en A* que buscan el camino más corto a partir de heurísticas basadas en distancias euclidianas [3] [15]. Por lo tanto, lo que nos interesa a nosotros es incluir, de alguna manera, la información **táctica** en esta heurística, tal y como lo hace por ejemplo el título Killzone [16].

La información táctica puede resultar un término un tanto ambiguo. Lo que en un juego puede considerarse información táctica, puede que en otro no lo sea. En general, la información táctica es un conjunto de datos, generalmente relacionados con el entorno, que nos ayuda a tomar una decisión mejor a la hora de realizar una acción, sea la que sea, atacar, cubrirnos, recargar o en este caso simplemente movernos.

La decisión de qué datos determinan la información táctica de un agente está a cargo del diseñador. La mayoría de veces esta hace referencia a la presencia de lineas de fuego amigas o enemigas, cercanía de coberturas o cercanía de posiciones de ataque, entre otros. Cabe destacar que la obtención de esta información táctica no es ni mucho menos trivial y que es en si misma otro reto al cual los diseñadores y programadores debemos enfrentarnos.

Una vez obtenida y procesada esta información, debemos incluirla en el sistema de costes de nuestro algoritmo de *pathfinding*. En muchas ocasiones, no resulta una tarea fácil, ya que al combinarla con la distancia euclidiana se deben asignar pesos, los cuales son muy difíciles de ajustar. A veces, vale la pena hacer un camino mucho más corto aunque haya peligro, en vez de hacer un camino mucho más largo sin peligro alguno.

Definir y obtener la información táctica, así como combinarla de manera correcta con la distancia, es la clave para crear agentes que se desplacen de manera táctica, tal y como lo haría un jugador real.

2.4.3 Predicción del jugador

Otro de los puntos que avanzamos antes (ver apartado 1.2) era el de predecir la posición del jugador. Una gran parte de los sistemas son capaces de hacerlo en situaciones a corto plazo, es decir, cuando el tiempo transcurrido entre la última vez que vimos al jugador y el momento de hacer la predicción son bajos. Cuando estos tiempos superan los 5 ó 10 segundos, es necesaria una herramienta diferente y específica para que las predicciones sean acertadas.

Una de estas herramientas que más se ha utilizado en la ultima década son los

mapas de influencia [17]. Esta técnica se ha utilizado principalmente en juegos de estrategia y en estos últimos años ha ganado popularidad en los juegos de tipo *FPS*, como el KillZone [18].

En general, los mapas de influencia se utilizan para ayudar a la IA a tomar mejores decisiones a partir del uso de información del entorno. Aunque se pueden utilizar con otros fines, como para guardar información táctica o información estadística histórica, uno de sus usos más comunes es el de hacer predicciones.

Su funcionamiento es muy simple. Están formados por una estructura de datos principal que representa el mapa navegable del nivel del videojuego (normalmente en forma de grafo o matriz) y donde a cada nodo o posición le corresponde un valor de influencia. Esta influencia se propaga a través del tiempo y del espacio por los vecinos de cada nodo o posición (ver Figura 2.12).

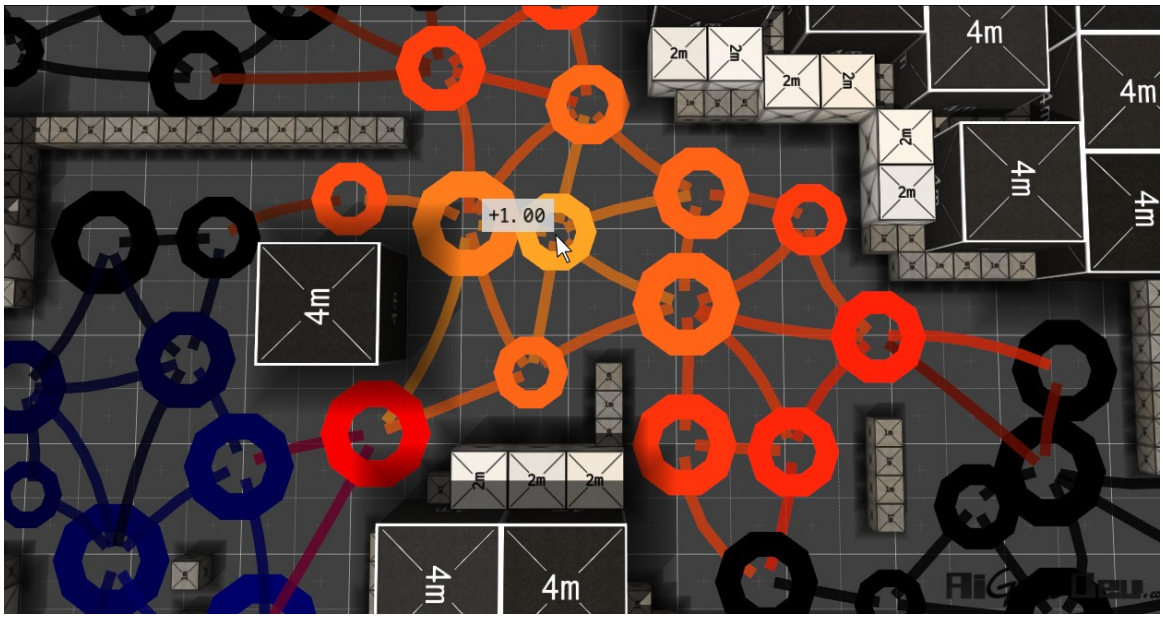


Figura 2.12: Propagación de la influencia a partir de un nodo con 1 de influencia.

En el caso que nos atañe, simplemente tendríamos que asignar una influencia a la ultima posición en la cual hemos visto al jugador para que esta se propague a medida que pasa el tiempo, y saber en todo momento cual es la posición más probable en la que se encuentre el jugador.

No obstante, como ya veremos más adelante en el apartado 6.4 hay diferentes maneras de configurar un mapa de influencia dependiendo de su configuración. Por este motivo es importante iterar y trabajar con diferentes configuraciones hasta encontrar la que más se ajusta a nuestro sistema.

2.4.4 Razonamiento espacial dinámico

Como hemos visto anteriormente, la fase de “Pensar” o “Razonar” de los agentes

inteligentes se encarga de procesar los datos e información captados durante la fase de “Sentir” para tomar decisiones. Cuantos más datos conocemos sobre el entorno, mejores decisiones podemos tomar.

El razonamiento espacial es razonar y posteriormente actuar acorde al entorno en el que se encuentra el agente. En los casos en los que el entorno o la situación pueden cambiar de forma drástica de un momento a otro, nos referiremos a esta capacidad de tomar decisiones de manera rápida basada en el entorno como razonamiento espacial dinámico.

Para que la toma de decisiones sea acertada es imprescindible la captación de grandes cantidades de información de manera rápida y eficiente. La mayor parte de esta información corresponde, dada la naturaleza de este tipo de juegos, a consultas de “Linea de Visión” (*Line-of-Sight* o simplemente *LoS*). Estas consultas se utilizan para obtener información relativa a la visibilidad de un objeto, como por ejemplo, si desde mi posición soy capaz de ver al jugador, si una posición es una buena cobertura o una buena posición de ataque, etc (ver 2.13).

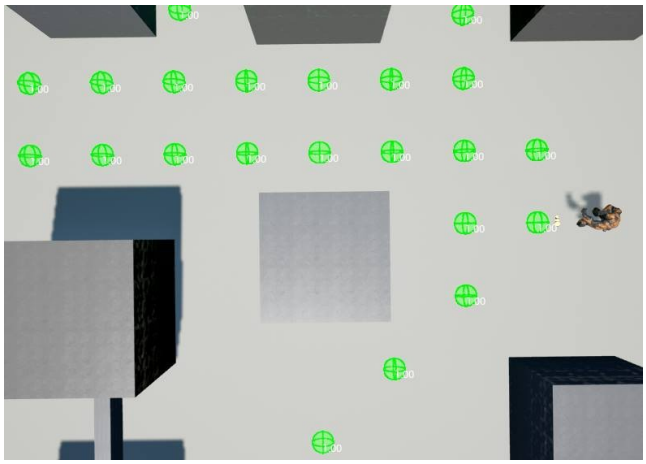


Figura 2.13: Posiciones visibles por un agente

Estas consultas son tan extremadamente comunes y necesarias debido a que podemos tener un entorno muy dinámico, muchos jugadores y/o muchos enemigos. Es importante que controlemos su uso, ya que cada una de estas consultas tiene un coste considerable, que si multiplicamos por la gran cantidad que realizamos, se vuelve mucho mayor aun, impidiendo la correcta ejecución del sistema.

Por este motivo, diversas soluciones con diferentes aproximaciones se han utilizando en los últimos años. La más popular es el uso de anotaciones o *waypoints.* Esta técnica se basa en incluir elementos (invisibles para los jugadores) en el juego con el objetivo de acelerar los cálculos. Esto es posible por dos motivos:

- **Espacio discreteado.** Al incluir anotaciones en el juego, el espacio que tenemos que evaluar a la hora de hacer los cálculos se reduce

considerablemente. Por ejemplo, cuando hacemos cálculos relativos a cobertura solo tendremos en cuenta las anotaciones de cobertura y no todo el espacio del nivel del juego.

- **Información pre-calculada.** Cada una de estas anotaciones es capaz de guardar información extra, calculada *offline.*

El uso de esta técnica (y otras) nos permite ahorrar o evitar cálculos que puedan afectar el rendimiento del juego. Normalmente, los diseñadores utilizan estas anotaciones para marcar zonas de cobertura, zonas de ataque o incluso ir más allá como hizo *Guerrilla Games* [19]. En Killzone 2, utilizaban los *waypoints* para guardar mapas de visibilidad pre-calculados, entre otras cosas, con el fin de evitar consultas de *LoS* en tiempo de ejecución (ver Figura 2.14).

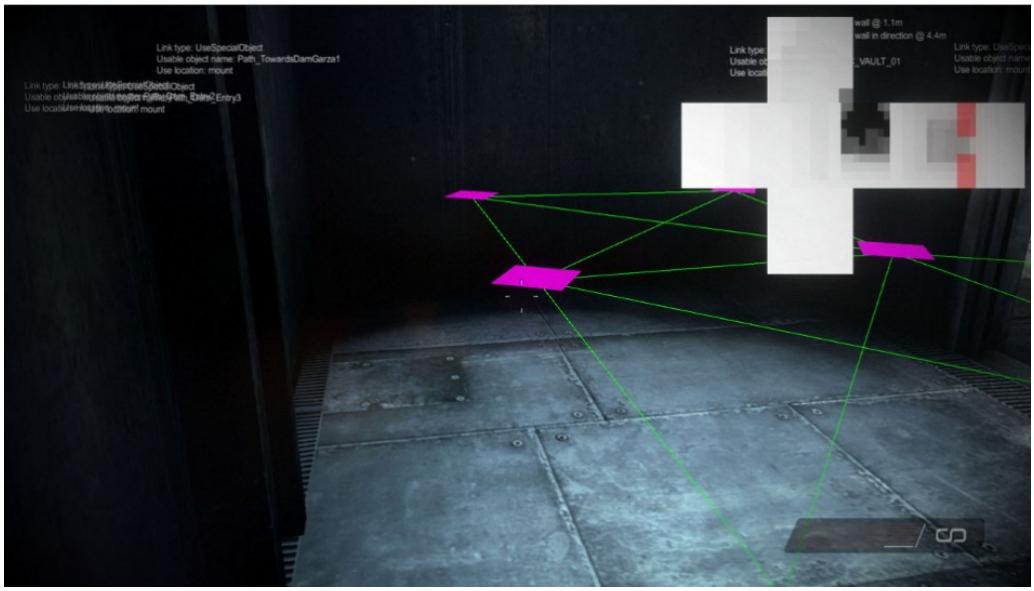


Figura 2.14: Killzone 2 utiliza waypoints con cubemaps (arriba a la derecha) pre- cálculados para determinar la visibilidad de cada zona.

Como hemos podido ver, el diseño de estos sistemas no sólo consiste en conocer y procesar la información que necesitamos para la toma de decisiones. También debemos saber gestionar su captura para que no afecte negativamente a la experiencia de juego.

2.4.5 Comportamientos colectivos

Por ultimo, uno de los retos pendientes y más complejos de la IA es el comportamiento en grupo. Es decir, que los agentes inteligente no trabajen de manera aislada, si no que se comuniquen y cooperen, haciendo uso de sus habilidades únicas para conseguir un objetivo común.

Años atrás, este comportamiento no era tan común debido a su elevada complejidad. Hoy en día, cada vez más y más videojuegos trabajan para incorporar

comportamientos grupales en sus sistemas de inteligencia artificial.

En la mayoría de ocasiones, juegos con comportamientos grupales (bien diseñados) generan mucha más diversión y, sobretodo, una necesidad de jugar sabiamente para vencer a un sistema completo que coopera para vencer al jugador. Esto resulta en experiencias más emocionantes, ya que incrementan enormemente el nivel de inmersión del jugador. Generalmente estos comportamientos grupales se utilizan en cualquier parte del sistema. Comúnmente, en los juegos *FPS* se utilizan para mejorar el *pathfinding* o las operaciones tácticas tanto de defensa como de ataque.

En el *pathfinding*, dependiendo de nuestro objetivo, nos puede interesar desplazarnos de una manera u otra. Supongamos que nuestro sistema de AI está inspeccionando un lugar desconocido. En este caso, probablemente nos interese que los agentes se desplacen manera muy compacta pero manteniendo una visión lo más amplia posible para evitar puntos ciegos, tal y como lo haría un comando de fuerzas especiales [15]. No obstante, si tenemos la necesidad de inspeccionar un área grande quizás nos interese que se separen y cubran el mayor terreno posible.

Cuando se trata de operaciones tácticas el problema se puede volver incluso más complejo. En situaciones de ataque, algunos agentes tienen que proporcionar fuego de supresión mientras otros avanzan hacia el enemigo o si se han quedado en una posición descubierta. De una manera similar, si un agente está en una situación desfavorecida (poca vida o munición) deben socorrerlo, tal y como haría un jugador humano.

Hemos podido ver que dependiendo de la situación y de las necesidades, el comportamiento grupal puede ser diferente. No obstante, en todos los casos, la filosofía que hay detrás es la misma. En ciertas ocasiones, el comportamiento grupal debe sobreponerse al comportamiento individual de los agentes, modificando así su conducta. Tener la capacidad de discernir entre el comportamiento individual o grupal es el autentico desafío [20], siendo una tarea difícil incluso para los humanos.

En la mayoría de casos, estos sistemas utilizan una arquitectura basada en capas, aunque que no tienen porque estar explícitamente definidas. Todas estas capas tienen siguen el ciclo “Sentir-Pensar-Actuar”, aunque con pequeñas diferencias.

Estos sistemas como mínimo están formados por dos capas: la capa inferior y la superior. La capa inferior corresponde a la inteligencia individual y la capa superior a la inteligencia colectiva de todos los agentes. Cada capa tiene un objetivo único que cumplir, siendo las capas superiores las más prioritarias.

No obstante, es muy común que estos sistemas incorporen capas intermedias. Estas capas intermedias, trabajan en un punto medio entre un único agente y el conjunto de todos los agentes (ver Figura 2.15: Esquema de un sistema de IA con múltiples capas).

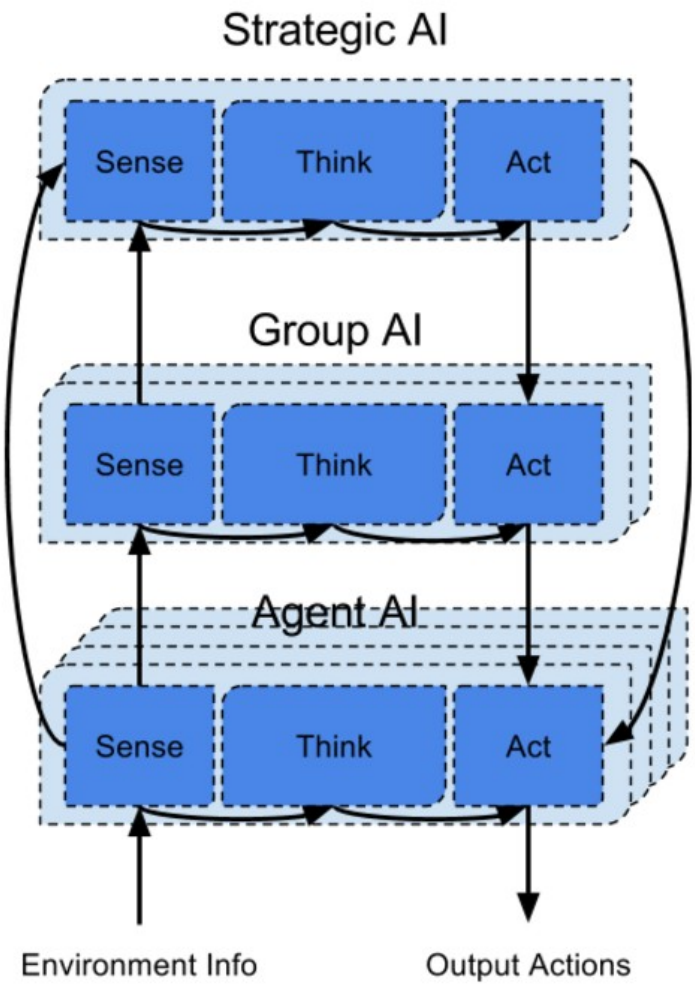


Figura 2.15: Esquema de un sistema de IA con múltiples capas

La capa inferior opera a nivel individual y es la encargada de captar los datos directamente del entorno a partir de los sensores (Sentir), ejecutar un proceso de toma de decisiones (Pensar) y ejecutar una acción (Actuar).

La o las capas intermedias, como hemos dicho, suelen trabajar a un nivel grupal que suele operar sobre un sub-conjunto pequeño respecto del total de los agentes. Dado que esta capa no puede captar datos directamente del entorno, los obtiene a partir de la capa inferior (la individual). Mediante el análisis de los datos de los diferentes agentes que controla, lleva a cabo un proceso de toma de decisiones y envía el resultado (las acciones o conjunto de acciones) a la inteligencia individual de los diferentes agentes, para que actúen.

La capa superior junta los datos de todos los agentes y/o de las diferentes capas, los procesa nuevamente a un nivel más alto y lleva a cabo otro proceso de toma de decisiones para acabar enviado un conjunto de acciones u órdenes a la capa

inferior, o directamente a los agentes, para que actúen.

A medida que pasamos a capa o capas superiores, la información se encuentra a un nivel más alto, y por lo tanto más abstracto. De una manera contraria, cuando el resultado de la toma de decisiones baja capa a capa, se va descomponiendo en tareas o acciones a nivel cada vez más bajo hasta que llega a la inteligencia individual, transformada en un conjunto o secuencia de acciones ejecutables por el agente.

Un ejemplo de esta arquitectura es la utilizada por *Killzone 2* (ver Figura 2.16) [21].

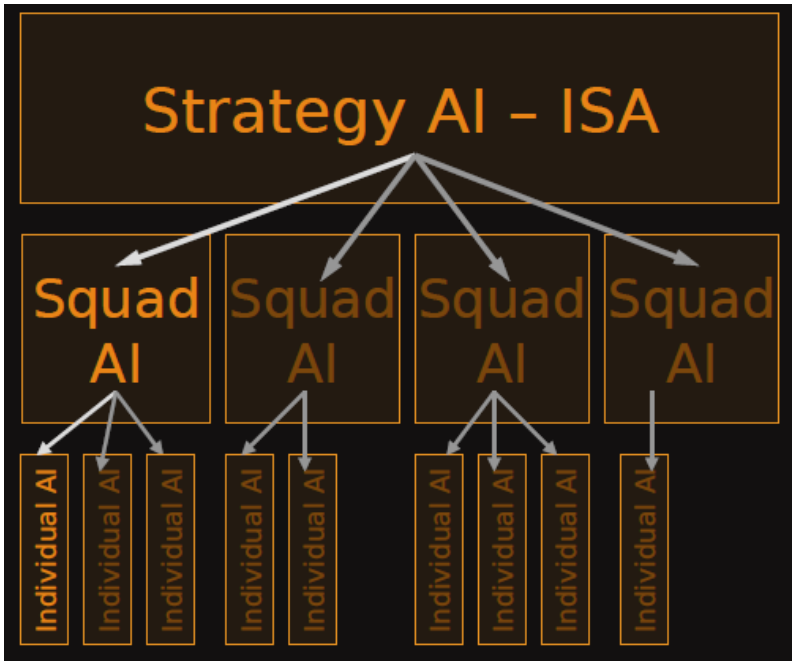


Figura 2.16: Sistema de inteligencia por capas de Killzone 2

En esta arquitectura ellos diferencias tres tipos de interacciones entre las capas:

- **Interacciones Strategy y Squad.**
  - Strategy → Squad. Ordenes a alto nivel como atacar, defender, flanquear, etc.
  - Squad → Strategy. Feedback relacionado con las ordenes. Por ejemplo orden realizada con éxito o fallada.
- **Interacciones Squad e Individual.**
  - Squad → Individual**.** Ordenes de desplazamiento.
  - Individual → Squad. Información relativa al entorno y al combate.
- **Interacciones Strategy e Individual.**

  Strategy → Individual. Ordenes de asesinato: Eliminar un objetivo

concreto.

- Individual → Strategy. Petición de resignación de tareas.

Estos sistemas probablemente sean los más complejos de los que hemos tratado hasta ahora, pero también son los que más valor pueden aportar a un juego. Por este motivo, cada vez vemos más y más juegos con sistemas de inteligencia colectivos.

03

Entornos de trabajo analizados

3 Entornos de trabajo analizados

3.1 Introducción

Decidir las tecnologías y herramientas que utilizaremos, así como el juego base, es probablemente la decisión más importante y que más puede influir en el desarrollo del proyecto. En cierta manera, escoger unas tecnologías u otras limita a la hora del desarrollo. Sobra decir que también determinan forzosamente el juego base que utilizaremos como punto de partida.

En los últimos años, en el mundo de los videojuegos, a este conjunto de tecnologías y herramientas que facilitan el desarrollo de los videojuegos se las conoce como **motores**. Actualmente existen diversos motores, algunos privativos como Anvil de Ubisoft y otros gratuitos como Unity e incluso de código libre como Unreal Engine. Cada uno esta definido por las herramientas que incorpora que lo hace poseer unas características únicas y por tanto, unas finalidades concretas.

No obstante, también existen alternativas a los motores gráficos, como desarrollar los videojuegos unicamente desde *IDEs (Integrated Development Environment)* aunque cada vez son más inusuales a causa de su elevada complejidad en comparación con los motores gráficos. Por este motivo, esta es una alternativa que fue descartada inmediatamente.

En este aparado, analizaremos los diferentes motores para identificar aquel que se adapta mejor a nuestros requisitos (explicado en el siguiente apartado Requisitos). El más importante sin duda es la existencia de un juego de tipo *FPS* sobre el cual desarrollar el proyecto.

3.2 Requisitos

Como ya avanzamos previamente, para elegir con responsabilidad y consciencia el motor que mejor resultado nos pudiese dar, elaboramos una lista de requisitos. Esta lista contiene las herramientas o características que consideramos relevantes para que el proyecto se desarrolle de manera correcta.

La lista, en orden descendiente de prioridad, es la siguiente:

- **Herramientas de IA.** Obviamente, es imprescindible que el motor incorpore herramientas específicas para apoyar el desarrollo de los sistemas de IA. Por ejemplo, sistemas de percepción del entorno, sistemas de toma de decisiones, etc.
- **Juego base**. También, totalmente indispensable, es que el motor disponga de un juego base (demo, tutorial, juego libre, etc) de tipo *FPS* que sea fácilmente accesible para crear su sistema de IA.

- **Curva de aprendizaje**. Dado que la duración del proyecto es corta, es importante evaluar el tiempo que tardare en acostumbrarme y a usar el motor con fluidez.
- **Comunidad y documentación**. Estrechamente relacionado con el apartado anterior, es importante que detrás del motor haya una comunidad dispuesta a ayudar, así como una documentación extensa y de buena calidad para agilizar el aprendizaje.

3.3 Opciones analizadas

Elaborada la lista de requisitos, el siguiente paso es evaluar como responde cada uno de los motores más populares y gratuitos a nuestros requisitos.

3.3.1 *Source 2*

3.3.1.1 *Introducción*

Source 2, aunque puede no ser un motor tan conocido como sus competidores, está detrás de títulos tan populares como *Dota 2, Portal 2, Left 4 Dead, Team Fortress 2* entre otros. *Source 2* es el motor desarrollado por y para Valve, y está íntimamente ligado a *Steam* por lo que su uso está muy extendido [22].

3.3.1.2 Evaluación de requisitos

A diferencia de otros motores que veremos más adelante como *Unity* y *Unreal* con los que ya tenia experiencia previa, *Source* era totalmente desconocido para mi. No obstante, decidí probarlo.

La verdad es que el aspecto visual del entorno no esta muy cuidado y resulta poco intuitivo y difícil de aprender. La presencia de una comunidad, tutoriales o documentación es totalmente inexistente. Aparentemente, aunque el motor sea gratuito, nadie fuera de *Valve* lo utiliza.

Aunque sí que encontramos un juego base muy atractivo que funcionaba sobre Source: *Alien Swarm.* Un juego de tipo *shooter* de código abierto que consiste, de manera resumida, en matar aliens. Tras inspeccionarlo detenidamente vimos que que era difícil realizar cualquier tipo de modificación. No conseguimos acostumbrarnos al entorno de desarrollo de *Valve.*

Por todos estos motivos, que sumados provocaban una expectativa de aprendizaje y progresión del proyecto preocupantemente lenta, descartamos *Source 2* como motor gráfico principal para este proyecto.

3.3.2 *CryEngine*

3.3.2.1 *Introducción*

Otro de los a veces olvidados motores, es *Cry Engine.* Desarrollado en 2005 por *CryTek* y por primera utilizando en el increíble titulo FarCry, ha ido ganando adeptos a lo largo de estos años, como *Crysis* y *Sniper: Ghost Warrior 2,* entre otros *[23].*

CryEngine esta orientado principalmente a profesionales y al desarrollo de juegos *AAA*[1]*.* Por estos motivos, se caracteriza por incorporar un conjunto de herramientas muy completo, aunque también complejo, y por una calidad asombrosa en comparación con otros motores.

No resulta extraño entonces, que *Amazon* haya creado su propio motor AAA llamado *Amazon Lumberyard* basado en *CryEngine* tras haber comprado las licencias de este último motor por más de 50 millones de dolares [25].

3.3.2.2 *Evaluación de requisitos*

Leída la definición inicial, ya podemos percibir que *CryEngine* es un motor que, aunque incorpora potentes herramientas de diferentes ámbitos incluida la IA, es demasiado complejo. La inversión de tiempo necesaria que se debería hacer para aprender a utilizarlo antes de poder desarrollar un sistema completo de IA es demasiado grande y dada la duración del proyecto no se llegaría a amortizarla.

Este proyecto no pretende ser un AAA por lo que este motor fue descartado de manera rotunda, aunque probarlo fue toda una experiencia.

3.3.3 *Unity 3D (Unity 5)*

3.3.3.1 *Introducción*

Unity 3D (*Unity* 5) es junto a *Unreal Engine 4* (ver Unreal Engine 4) uno de los gigantes en el mundo de los motores gráficos. Los principales motivos son su accesibilidad por parte de desarrolladores y estudios independientes, su habilidad para desarrollo multiplataforma, pero sobretodo la facilidad a la hora de utilizarlo. Prácticamente cualquier persona, aunque no posea grandes conocimientos informáticos, puede ser capaz de desarrollar un juego con *Unity*. Por esta razón, desde el 2004, año de su primer lanzamiento, *Unity* ha evolucionado mucho hasta convertirse en el motor gráfico con la comunidad *online* más grande. Incluso la multinacional *Blizzard* utiliza *Unity* para desarrollar si ultimo videojuego: Hearthstone: Heroes of Wacraft (2016) [26].

1 Videojuegos mejores vendidos o de muy alta calidad comúnmente con elevados presupuestos de desarrollo *[24]*.

3.3.3.2 *Evaluación de requisitos*

La ventaja de *Unity* sobre los otros dos motores, es que ya poseíamos una experiencia previa con esta herramienta ya que fue la utilizado para desarrollar dos videojuegos durante el máster.

Como ya comentamos anteriormente, *Unity* es muy fácil de utilizar. La comunidad de desarrolladores que hay detrás es enorme y la documentación y tutoriales son de muy buena calidad. Pero, la mayoría de la gente no traba en profundidad los temas de IA, por lo que la documentación en esta ámbito es mucho menor.

No obstante, *Unity* no incorpora en si mismo herramientas de IA para modelar comportamientos como lo hacen los otros motores. Unicamente incorpora algoritmos de navegación. Si queremos por ejemplo, modelar el comportamiento de nuestros *NPCs,* tenemos dos opciones:

- Implementar la técnica nosotros, con lo que perderíamos más tiempo implementando el sistema que modelando el comportamiento.
- Utilizar plugins de terceros. En el mercado de *Unity* podemos encontrar, por ejemplo, implementaciones de árboles de comportamiento o máquinas de estado [27]. El principal inconveniente de estos plugins es que están desarrollados por gente ajena a Unity y por lo tanto, no nos garantizan ni su funcionamiento, ni su soporte.

Por lo que respecta al juego base, *Unity* dispone de un tutorial de un juego de disparos *Survival Shooter* bastante simple (ver Figura 3.1) el cual serviría como juego base *[28].*

Figura 3.1: Unity tutorial Survival Shooter

Podemos deducir entonces, que a diferencia de los motores anteriores (*Source 2* y *CryEngine)* que *Unity* podría ser una herramienta válida para el desarrollo del proyecto (ver tabla 2).

	Unity 3D
Herramientas de IA	No - Sólo plugins de terceros
Juego base	Survival Shooter – Muy simple
Curva de aprendizaje	Leve
Comunidad y documentación	Muy buena

Tabla 2: Evaluación de los requisitos necesarios por parte del motor Unity 3D

3.3.4 *Unreal Engine 4*

3.3.4.1 Introducción

Como avanzamos previamente, *Unreal Engine 4* es el gran competidor de *Unity 3D* aunque compiten en ligas diferentes. Mientras que *Unity* pretende ser un motor accesible y fácil de utilizar, orientado especialmente a pequeños estudios o desarrolladores independientes, *Unreal Engine* sigue una filosofía diferente. Aunque también puede ser utilizado por desarrolladores o estudios independientes, *Unreal* es un motor más complejo y difícil de utilizar. Por esta razón, su desarrollo esta más orientado hacia los profesionales, quienes buscan un motor con una calidad y unas herramientas de nivel superior, en una línea similar a *CryEngine.* Esto provoca, a diferencia de *Unity,* que su comunidad sea mucho más pequeña y su uso mucho más complejo, aunque la calidad de los resultados también son notablemente diferentes entre uno y otro.

Algunos de los títulos desarrollados con *Unreal Engine* son la clásica saga *Unreal Tournament, Deus Ex* o el tan esperando *remake* del *Final Fantasy VII entre otros [29].*

3.3.4.2 Evaluación de requisitos

Al igual que con *Unity*, ya habíamos trabajado previamente con *Unreal,* por lo se tenía unos sólidos conocimientos de su funcionamiento.

Unreal es un motor más complejo que *Unity* y por lo tanto, más difícil de utilizar y dominar. Esto provoca que su comunidad sea más pequeña y su documentación no tan extensa.

Sin embargo, esta complejidad viene dada a que *Unreal* esta dotado de herramientas mucho más avanzadas que *Unity.* Incorpora diversas utilidades que

ayudan notablemente en el desarrollo de la IA, como por ejemplo: navegación, árboles de comportamiento, sistemas de percepción, sistemas de consulta sobre el entorno (*Environment Query System o EQS),* etc (ver tabla 3).

	Unreal Engine 4
Herramientas de IA	Si (Pathfinding, BT, EQS, Perception...)
Juego base	Shooter Tutorial – Muy Completo
Curva de aprendizaje	Pronunciada
Comunidad y documentación	Mediocre

Tabla 3: Evaluación de los requisitos necesarios por parte del motor Unreal Engine 4

Ademas, uno de los tutoriales de Unreal es precisamente un *FPS* muy completo y mucho más atractivo que el de *Unity* (ver Figura 3.2).

Figura 3.2: UE4 tutorial FPS

3.4 Opción escogida

Entre las diferentes opciones evaluadas, recordemos que descartamos *CryEngine* y *Source 2* por diversos motivos, pero principalmente por falta de experiencia con ellos y por la gran dificultar que conlleva aprender a utilizarlos. Los candidatos restantes son *Unity 3D* y *Unreal Engine 4.*

	Unity 3D	Unreal Engine 4
Herramientas de IA	No - Sólo plugins de terceros	Si (Pathfinding, BT, EQS, Perception...)
Juego base	Survival Shooter – Muy simple	Shooter Tutorial – Muy Completo
Curva de aprendizaje	Leve	Pronunciada
Comunidad y documentación	Muy buena	Mediocre

Tabla 4: Comparativa de las características necesarias para el proyecto de Unity 3D y Unreal Engine 4.

En la tabla 4 podemos ver que, mientras *Unreal* incorpora herramientas de IA de forma nativa, *Unity* no lo hace. Además, el juego base que encontramos en *Unreal* es más completo y atractivo que el de *Unity.* No obstante, el tiempo necesario para acostumbrarme a usar *Unity* es mucho menor que *Unreal* gracias a su simple funcionamiento y gran comunidad y documentación.

Finalmente, dado que *Unreal* responde satisfactoriamente a los dos requisitos de máxima prioridad y a que, consideramos que gracias a los conocimientos previos con el motor, será posible desarrollar el sistema de IA en el tiempo establecido. Hemos decidido que sea *Unreal Engine 4* el motor gráfico sobre el cual desarrollaremos la tesis.

04

Descripción del entorno de trabajo

4 Descripción del entorno de trabajo

4.1 Introducción

Una vez tomada la decisión de utilizar *Unreal Engine 4,* vale la pena dedicar un momento a analizar detenidamente las herramientas que ofrece y que podemos utilizar para construir el sistema de IA.

También estudiaremos el juego base para entender su funcionamiento, incluyendo su sistema de IA inicial para evaluar que posibilidades nos brinda. Es importante, antes de hacer cualquier modificación entender como encajan y funcionan los diferentes componentes del juego y de los *NPCs.*

4.2 Unreal Engine 4

4.2.1 Sistema de percepción

Como su mismo nombre indica, este sistema permite dotar a los NPCs con capacidades de percepción. Aunque aún no existe documentación oficial sobre cómo configurar los sistemas de percepción en *Unreal*, estos ya están incorporados en el motor y algunas personas explican como utilizarlos *[30].*

El uso de estos sistemas permite a los *NPCs* escuchar ruidos e identificar su procedencia, ver a enemigos y a aliados, detectar daños y el causante, etc. Esto resulta extremadamente útil para obtener información en la fase de “Sentir” del agente (ver Figura 4.1).

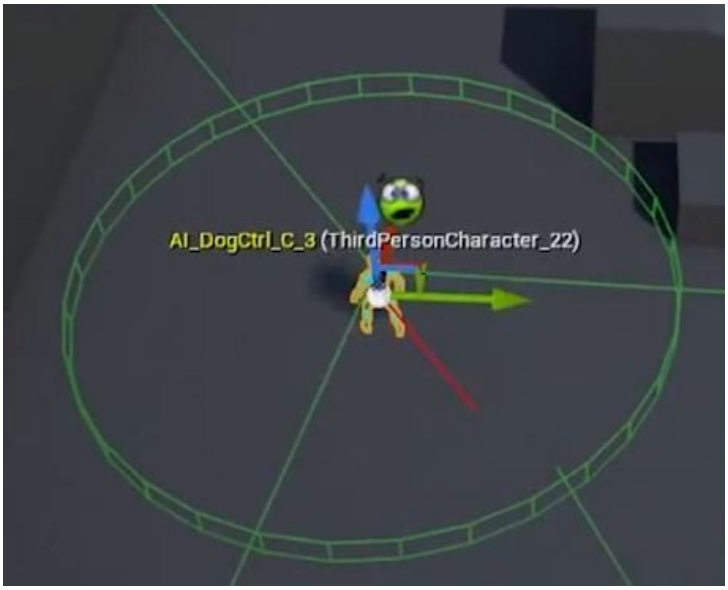


Figura 4.1: Sistema de percepción de sonido con un radio determinado en un NPC

4.2.2 Sistema de navegación

La función principal del sistema de navegación de *Unreal* es la generación de la malla de navegación (*Navigation Mesh,* o simplemente *Navmesh) [31]*. Esta malla define el terreno o las partes del mapa por las que los *NPCs* pueden desplazarse (ver Figura 4.2).

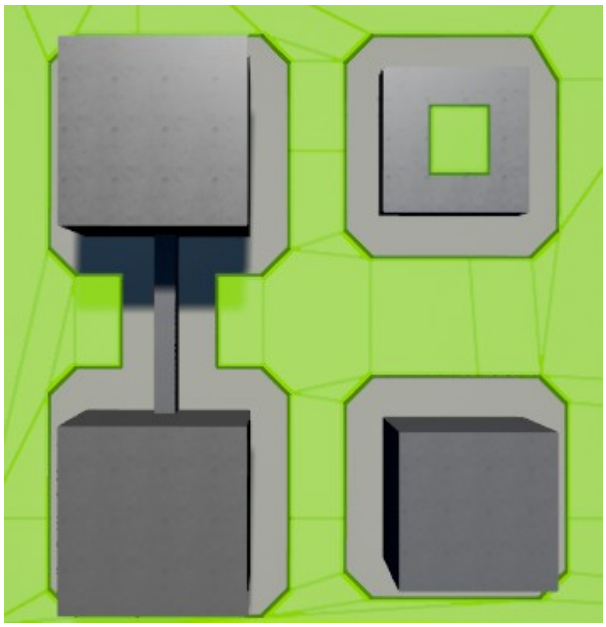


Figura 4.2: Navmesh generada por UE4 (en color verde)

El sistema de navegación de *UE4* es uno de los más completos. Además de permitir una alta personalización en la etapa de generación la *Navmesh,* permite que esta sea estática o dinámica. Es decir, que nos permite trabajar con una malla que se re-calcula en tiempo de ejecución para crear entornos más dinámicos y experiencias más vivas. Asimismo, también nos permite crear lo que ellos definen como enlaces de navegación (*Nav Link Proxy).* Estos enlaces permiten a los *NPCs* abandonar temporalmente la *Navmesh* y saltar agujeros (ver Figura 4.3).

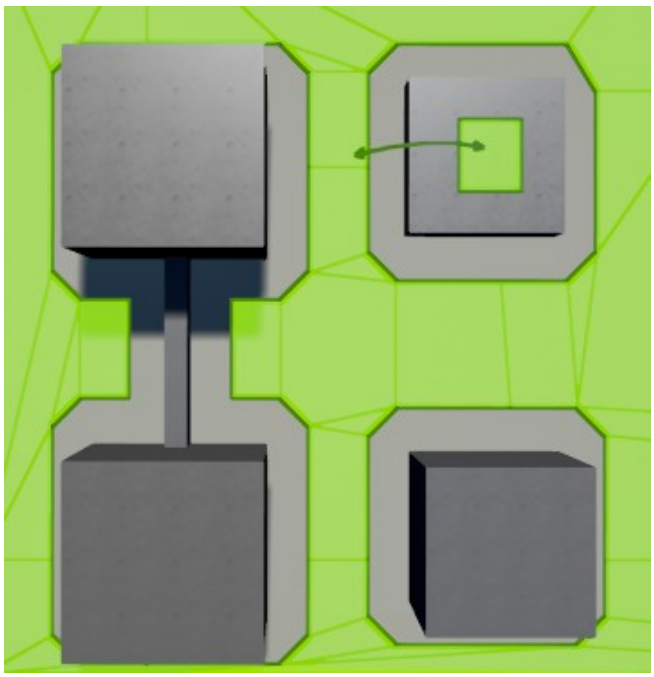


Figura 4.3: Navmesh generada por UE4 (en color verde) incluyendo un enlace navegable

Estos enlaces son utilizados como si fueran caminos ordinarios en las tareas de *pathfinding. Unreal* utiliza internamente un algoritmo *A** con una heurística basada en distáncia euclídea. No obstante, este coste se puede modificar de manera fácil utilizando *Nav Modifier Components,* los cuales te permite modificar el coste de una cierta área (ver Figura 4.4) *[32].*

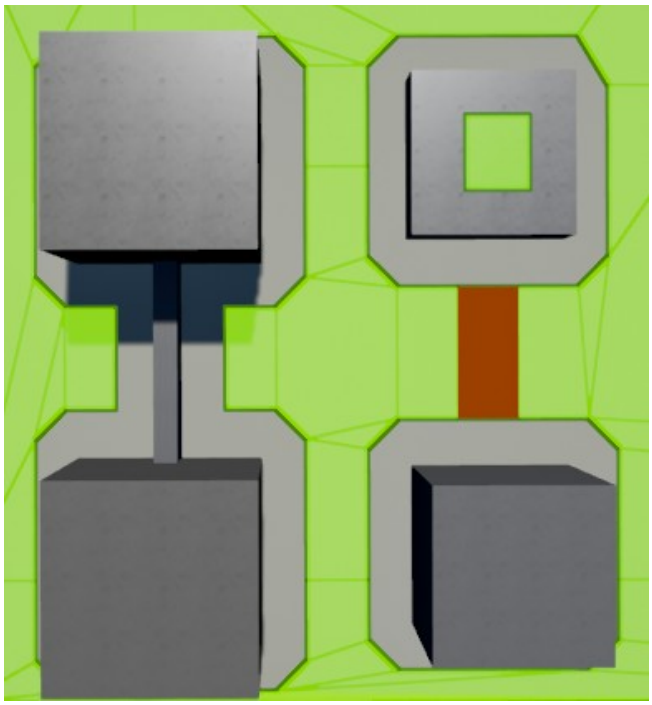


Figura 4.4: Navmesh generada por UE4 (en color verde) incluyendo un incrementador de coste de manera manual (en color rojo)

4.2.3 Behavior Trees

Como hemos visto, el hecho de que *Unreal* incorporara esta herramienta fue decisivo a la hora de tomar la decisión de qué motor utilizar. Como ya explicamos previamente el funcionamiento de los árboles de comportamiento (ver apartado 2.3.3), ahora nos centraremos en como funcionan en este motor gráfico *[33].*

Lo más importante es que *Unreal* ha decidido separar lo que es el comportamiento del *NPC* de su memoria. Para ello, la memoria se configura en un componente separado llamado *Blackboard* y el comportamiento en un componente *Behavior Tree.* Esto permite abstraer al árbol de la estructura de los datos.

Para el diseño del árbol de comportamiento, *Unreal* incorpora una herramienta visual muy fácil de utilizar que permite construir el árbol agregando diferentes nodos. Estos nodos pueden ser de dos tipos: compuestos o tareas (ver Figura 4.5).

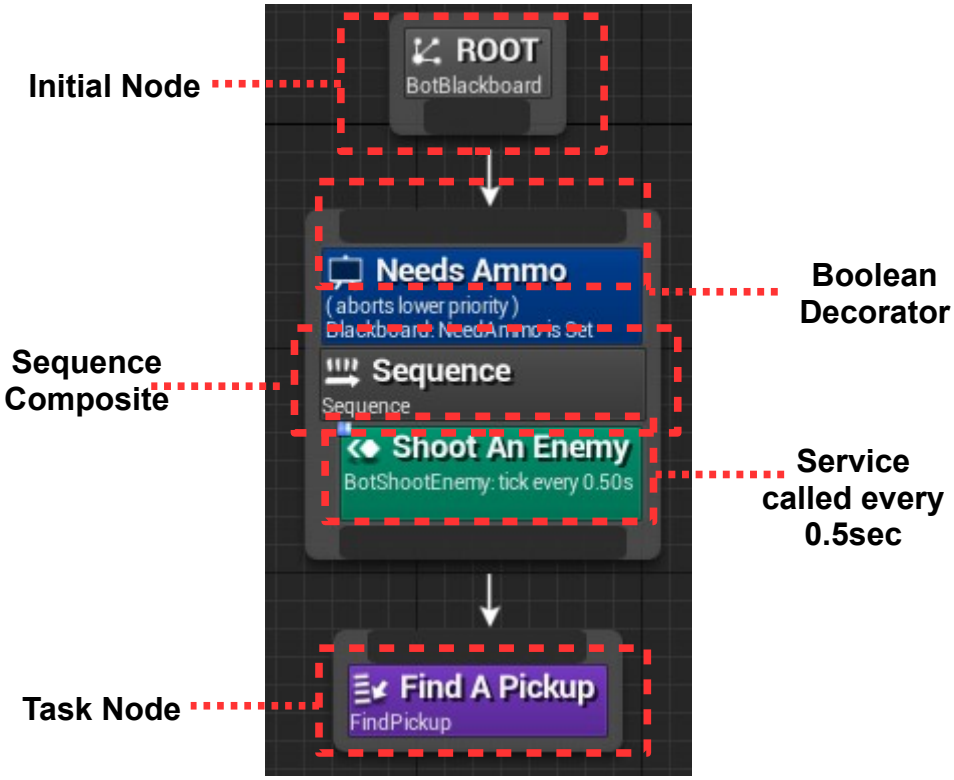


Figura 4.5: Herramienta visual para diseñar Behavior Trees en UE4.

Los nodos compuestos, tal y como explicamos antes, pueden ser secuencias, selectores o nodos paralelos.

Por lo que respecta a las tareas, simplemente son los nodos hojas que provocan que el *NPC* ejecute algún tipo de acción. Podemos crear tantas tareas como deseemos.

Unreal incorpora una herramienta que difiere de la explicación que hicimos de los arboles de comportamiento. Esta herramienta permite añadir a cada nodo un numero ilimitado de *Servicios*. Un servicio es una función que se llama de manera periódica, cada cierto tiempo determinado, mientras su nodo está activo. Un ejemplo claro sería el de hacer una actualización de los datos.

Por ultimo, estos *BT* también incorporan el uso de decoradores. Cada nodo puede tener un número ilimitado de decoradores. Estos pueden ser de diferentes tipos o incluso definidos por el diseñador. Además, permite configurar observadores en los decoradores para indicar cuándo abortar. Supongamos que tenemos una rama del árbol de comportamiento con un decorador de tipo booleano. Es decir, que sólo se ejecutará cuando la condición del decorador sea cierta. Si también le agregamos un observador, podemos forzar a que, una vez se este ejecutando esta rama del árbol, se aborte tan pronto como la condición del decorador cambie.

4.2.4 *Environment Query System (EQS)*

Esta funcionalidad, por ahora experimental, permite a los *NPCs* hacer preguntas sobre el entorno para recolectar información *[34].* Tal y como podemos pensar, esta herramienta, al igual que los sistemas de percepción, nos ayudaran enormemente durante la fase de “Sentir” del agente.

Su funcionamiento es muy simple y fácilmente extensible:

1. Un elemento de tipo *Generador* genera un junto de elementos de acuerdo a unos condiciones que le hayamos indicado y en referencia a un elemento determinado. Normalmente, suele generar una rejilla o circulo de un radio determinado alrededor del *NPC.*
2. Posteriormente, cada uno de los elementos se somete a diferentes tests. Los tests discriminan los elementos buenos de los malos, filtrándolos, aplicándoles una cierta puntuación o ambas cosas. Por ejemplo, podríamos utilizar un test distancia que favoreciese los puntos cercanos sobre los lejanos. De hecho, podemos utilizar tantos tests como queramos. Como resultado obtenemos un conjunto de elementos que han pasado los tests y que con su puntuación nos indican que tan bien los han pasado (ver Figura 4.6).

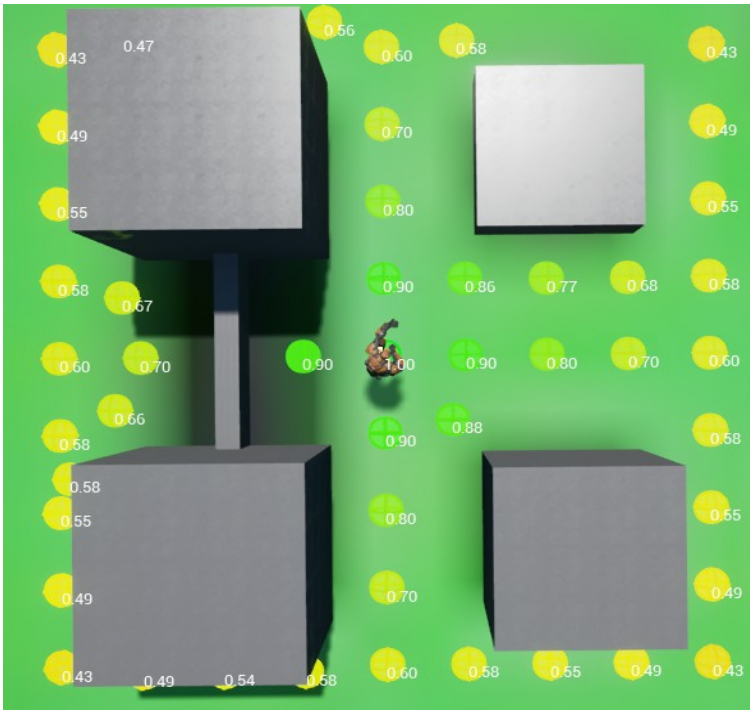


Figura 4.6: Elementos generados por una consulta EQS con forma de rejilla y un único test de distancia que favorece los elementos cercanos. El número al costado de cada elemento es su puntuación (más grande mejor)

4.3 Juego base

4.3.1 Sistema de IA inicial

El sistema de IA inicial del juego era algo muy rudimentario, que aunque no tenia mucho sentido desde punto de vista de inteligencia, ilustraba correctamente el uso de las diferentes herramientas de *Unreal* para trabajar con IA.

4.3.1.1 Sentir

La información que utilizaba este sistema era muy pobre. Simplemente tenia en cuenta si necesitaba munición y la posición del jugador. De hecho, no utilizaba ningún tipo de percepción para obtener la información del jugador. Accedía directamente al código interno de *Unreal* para obtener una instancia del jugador y por lo tanto su posición. Lo viese, escuchase o no. El *NPC* siempre sabia donde estaba el jugador.

4.3.1.2 Pensar

Como ya comentamos, este árbol de comportamiento pecaba de simplista. Simplemente, como siempre conocía la posición del jugador, se movía directo hacia él disparando sin parar. Cuando se quedaba sin balas en el cargador recargaba, y cuando se quedaba completamente sin munición caminaba de manera aleatoria hasta atravesar un área de suministro de munición.

Este ejemplo me sirvió para entender el funcionamiento de los *BT,* pero no para guiar en cómo modelar un comportamiento inteligente.

4.3.1.3 Actuar

Por lo que respecta a las acciones, los *NPCs* son capaces de:

- **Disparar** a una localización o a un objeto en movimiento.
- **Recargar** en cualquier momento siempre que tengan balas suficientes.
- **Moverse** utilizando caminos más cortos dentro de la *Navmesh.*

Estas acciones, aunque parezcan pocas, son totalmente suficientes para poder diseñar un agente inteligente.

05

Sistema de comportamiento

5 Sistema de comportamiento

5.1 Introducción

A lo largo de este apartado describiremos con gran detalle como funcionan cada uno de los componentes del el sistema de IA.

Procederemos haciendo una explicación *Top-down* que no se corresponde exactamente al orden cronológico en como se ha desarrollado el proyecto, pero que consideramos que es la mejor manera de explicarlo.

Comenzaremos explicando como razona el agente (apartado 5.2*)* así como la información necesaria para llevar a cabo este razonamiento. Después veremos como obtener esta información (apartado 5.3) y por último que acciones u operaciones puede ejecutar el agente para modificar el entorno (apartado 5.4).

5.2 Pensar

Para modelar el razonamiento del agente y su comportamiento en *Unreal* se utilizó la previamente explicada herramienta visual de árboles de comportamiento (ver apartado 4.2.3).

Para llevar a cabo este diseño de una manera modular se diseñaron diferentes árboles que definen distintos comportamientos. De esta manera, además de ser más fácil de entender, también es mas fácil re-utilizar los componentes.

5.2.1 Árbol de comportamiento principal

5.2.1.1 Responsabilidad

Determinar qué comportamiento a alto nivel se adapta mejor a la situación actual. Puede elegir entre cuatro comportamientos: *Idle* (ver apartado 5.2.2), *Patrol* (ver apartado 5.2.3), *Search* (ver apartado 5.2.4) o *Fight* (ver apartado 5.2.5). Además, este árbol es el encargado de ejecutar el servicio responsable de obtener los datos del entorno (ver apartado 5.4.1.1).

5.2.1.2 Activación

Este árbol, al ser el principal, describe completamente el comportamiento del agente y se ejecuta siempre y cuando el agente este funcionando y exista (esté vivo).

5.2.1.3 Desactivación

El árbol sólo deja de funcionar cuando el agente muere.

5.2.1.4 *Memoria*

La memoria de este árbol esta formada por una única variable *State.* Esta variable es de tipo *Enum* y su valor puede ser *Idle, Patrol, Search* o *Fight.* Ademas, esta es una variable de ámbito global, es decir, que la comparten todas las instancias del árbol que se estén ejecutando (ver tabla 5).

	Dato	Ámbito
State	Enum	Global

Tabla 5: Memoria del árbol de comportamiento principal

5.2.1.5 *Funcionamiento*

Durante su ejecución, este árbol siempre tiene qué decidir que comportamiento ejecutar de los cuatro posibles: *Idle*, *Patrol*, *Search* o *Fight* (ver Figura 5.1). Para tomar esta decisión, lo hace a partir del valor de la variable *State.*

Más adelante veremos con detalle en qué circunstancias la variable *State* toma cada uno de los valores (ver apartado 5.3.2.1). Ademas, como ya avanzamos, vemos que cada 0.3s ejecuta el servicio de *Sense Data* para obtener la información del entorno (ver apartado 5.4.1.1).

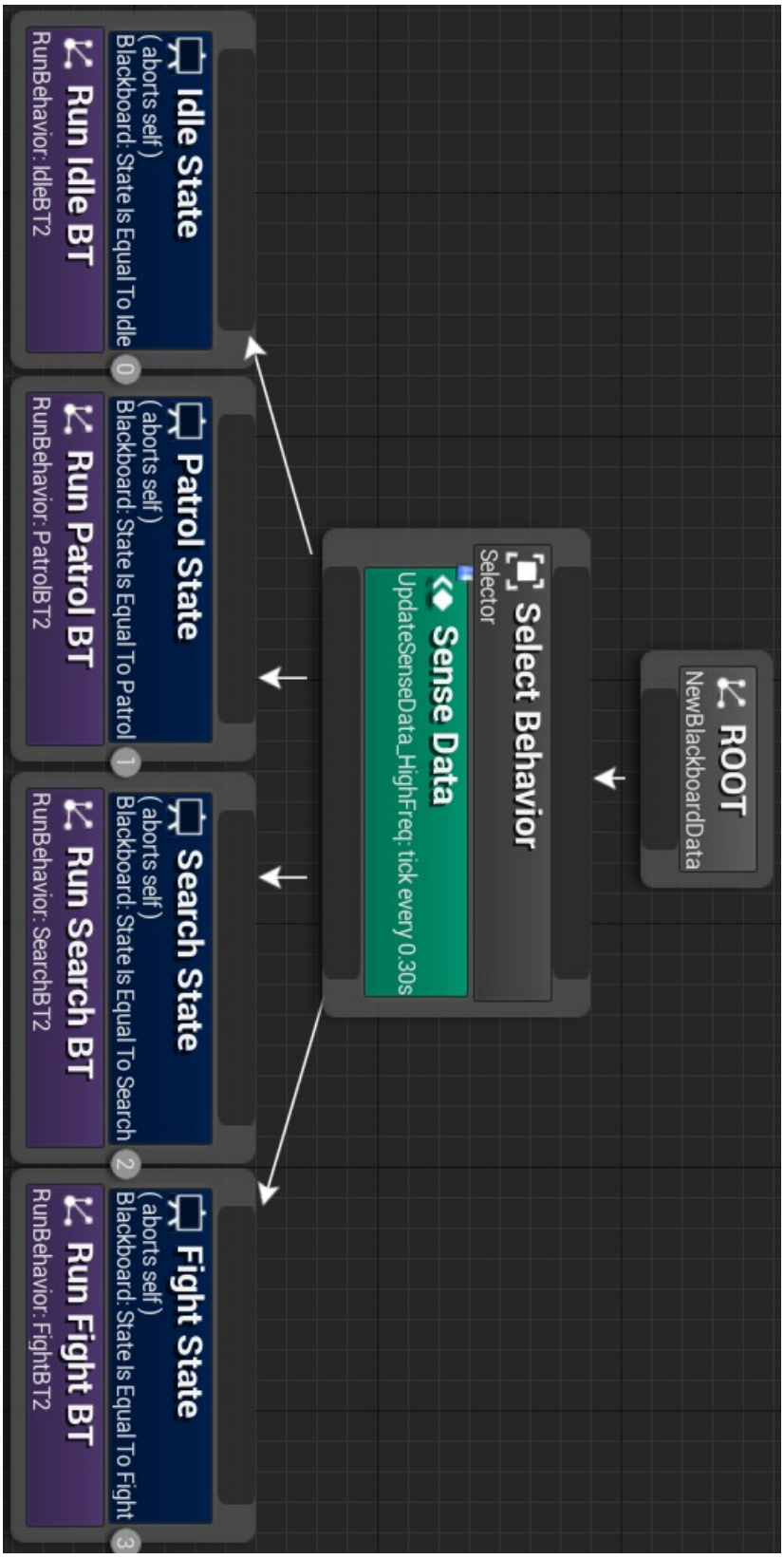


Figura 5.1: Árbol de comportamiento principal

5.2.2 *Árbol de comportamiento Idle*

5.2.2.1 Responsabilidad

Este comportamiento, tal y como su nombre indica, no hace nada.

5.2.2.2 Activación

Este comportamiento se ejecuta cuando el valor de la variable *State* es *Idle*. Como ya veremos más adelante en el apartado 5.3.2.1, este comportamiento únicamente se ejecuta cuando los *NPCs* han matado al jugador y por lo tanto han ganado la partida.

5.2.2.3 Desactivación

Cuando el valor de la variable *State* deja de ser *Idle,* el árbol acaba su ejecución, hayan o no finalizado las tareas que se estaban ejecutando.

5.2.2.4 Memoria

Dado que el comportamiento de este árbol es “nulo” no necesita guardar ningún tipo de información.

5.2.2.5 Funcionamiento

Como ya avanzamos este comportamiento es muy sencillo: mientras el agente se encuentra en el estado *Idle,* espera (ver Figura 5.2).

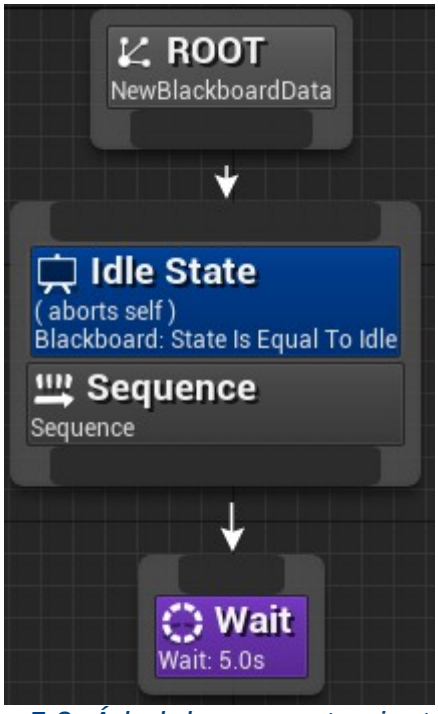


Figura 5.2: Árbol de comportamiento Idle

5.2.3 Árbol de comportamiento Patrol

5.2.3.1 Responsabilidad

Su propósito es simular como los *NPCs* protegen o defienden una zona patrullando a lo largo de una secuencia de puntos predefinida.

5.2.3.2 Activación

Es el estado inicial de los agentes y se ejecuta cuando el valor de la variable *State* es *Patrol.* Ya lo detallaremos más adelante, pero una vez los agentes salen de este estado nunca vuelven a él (ver apartado 5.3.2.1).

5.2.3.3 Desactivación

Al igual que antes, cuando el valor de la variable *State* deja de ser *Patrol,* el árbol finaliza su ejecución, hayan o no acabado las tareas que se estaban ejecutando.

5.2.3.4 Memoria

Este comportamiento es muy simple por lo que sólo necesitamos un par de variables: una que guarde todos los puntos de patrulla predefinidos y la otra que indique el siguiente punto de patrulla al que tiene que dirigirse el *NPC* (ver tabla 6).

	Dato	Ámbito
NextPatrolLocation	Vector	Individual
PatrolPoints	Array<Vector>	Individual

Tabla 6: Memoria del árbol de comportamiento Patrol

5.2.3.5 Funcionamiento

Su funcionamiento es muy simple (ver Figura 5.3). Mientras esta en el estado *Patrol* ejecuta una secuencia de tres operaciones:

0. Obtiene el siguiente punto de patrulla y lo asigna a la variable *NextPatrolLocation* (ver apartado 5.4.2.4.1)
1. Se desplaza al siguiente punto de patrulla *NextPatrolLocation* por el camino más corto (ver apartado 5.4.2.3)
2. Espera un tiempo aleatorio (ver apartado 5.4.2.2)

Esta secuencia se ejecuta de manera infinita mientras se encuentre en este estado. De esta manera, al finalizar una secuencia, el agente la vuelve a repetir pero con el siguiente punto de patrulla y así sucesivamente.

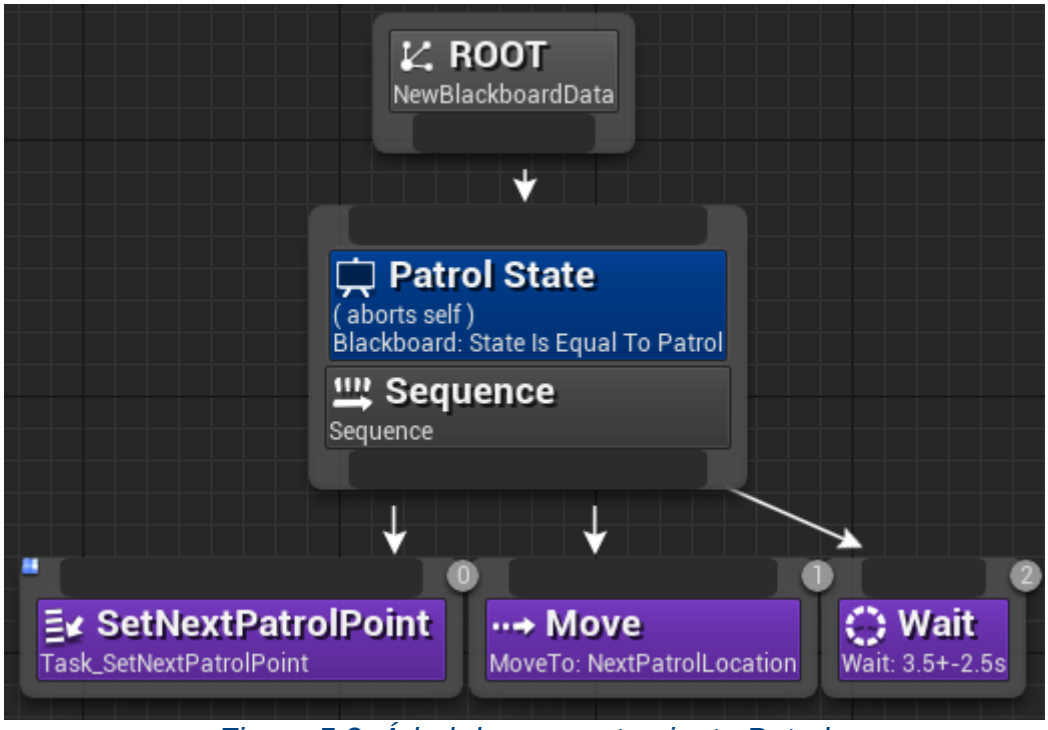


Figura 5.3: Árbol de comportamiento Patrol

5.2.4 *Árbol de comportamiento Search*

5.2.4.1 Responsabilidad

El objetivo de este comportamiento es que los agentes ejecuten una operación de búsqueda para encontrar al jugador.

5.2.4.2 Activación

Se ejecuta cuando el valor de la variable *State* es *Search.* Ya veremos más adelante que este comportamiento se ejecuta siempre que, una vez visto el jugador, perdamos la pista de donde se encuentra (ver 5.3.2.1).

5.2.4.3 Desactivación

Cuando el valor de la variable *State* deja de ser *Search,* el árbol acaba su ejecución, hayan o no finalizado las tareas que se estén ejecutando.

5.2.4.4 Memoria

De una manera similar al árbol anterior, este comportamiento necesita guardar el siguiente punto de búsqueda al que se dirigirá, *NextSearchPoint,* y también se usan dos estructuras auxiliares (ver tabla 7):

- Un único *PredictionMap* utilizado por todos los agentes para hacer predicciones y calcular las posiciones más probables en las que podría estar el jugador

- Un único *SearchMap* utilizado por todos los agentes para poner en común sus posiciones de búsqueda y así poder conocer la situación de los otros agentes y por lo tanto actuar de manera colectiva.

	Dato	Ámbito
NextSearchLocation	Vector	Individual
PredictionMap	Influence Map	Global
SearchMap	Map<String,Vector>	Global

Tabla 7: Memoria del árbol de comportamiento Search

5.2.4.5 *Funcionamiento*

El funcionamiento a alto nivel de este comportamiento es muy sencillo y también está formado por una secuencia de dos operaciones (ver Figura 5.4):

0. Calcula y asigna el siguiente punto de búsqueda *NextSearchLocation.* En el apartado 5.4.2.4.2 analizamos como se calcula este punto de búsqueda a partir del *PredictionMap* y del *SearchMap.*
1. Se desplaza a la localización *NextSearchLocation* utilizando el camino más corto (ver apartado 5.4.2.3).

Podemos ver que este comportamiento es conceptualmente muy similar al *Patrol* con la única diferencia que, en vez de movernos a un punto de patrulla, nos vemos a un punto en el que potencialmente podría estar el jugador. Error: Reference source not found

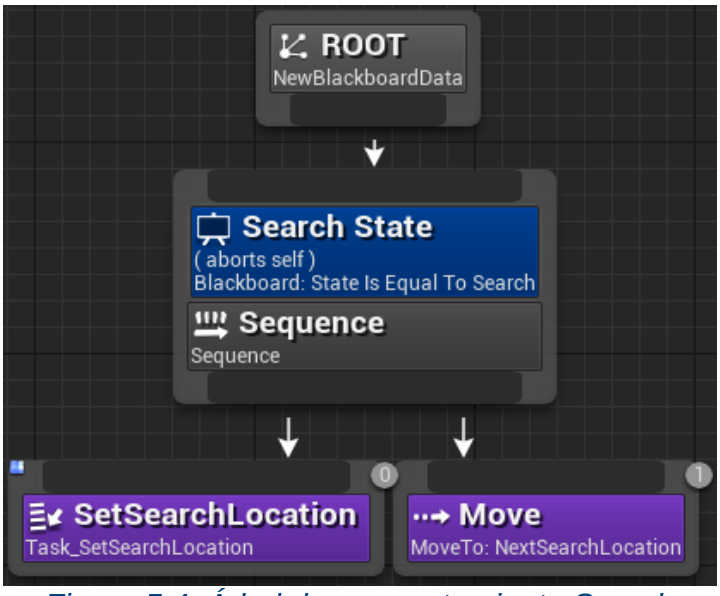


Figura 5.4: Árbol de comportamiento Search

5.2.5 *Árbol de comportamiento Fight*

5.2.5.1 Responsabilidad

Este es el comportamiento en el que los agentes pasaran la mayor parte de su tiempo y por esta razón es en el que más tiempo se ha invertido. Se trata del comportamiento de combate y su objetivo es simular cómo los agentes se defienden y atacan al jugador. Ademas, también es responsable de ejecutar el servicio encargado de disparar al jugador (ver apartado 5.4.1.2).

5.2.5.2 Activación

Se ejecuta cuando el valor de la variable *State* es *Fight.* Lo detallaremos más adelante, pero de manera muy resumida, los agentes se encuentran en un estado de combate siempre que ven al jugador o tienen una idea muy clara de donde se encuentra, en contraposición con el estado *Search,* en el cual no saben exactamente donde esta (ver apartado 5.3.2.1).

5.2.5.3 Desactivación

Al igual que en todos los casos anteriores cuando el valor de la variable *State* deja de ser Fight, el árbol acaba su ejecución, hayan o no finalizado las tareas que se estén ejecutando.

5.2.5.4 Memoria

Obviamente, como veremos a continuación, este comportamiento es el más complejo por lo que la información que utiliza para llevarse a cabo también lo es (ver tabla 8).

De manera similar a como pasaba en el estado *Search*, aquí tenemos una variable *AttackMap* compartida entre los agentes que sirve para poner en común las posiciones de ataque y realizar razonamientos más colectivos, como por ejemplo, atacar desde diferentes ángulos.

	Dato	Ámbito
ReloadSoon	Bool	Individual
ReloadNow	Bool	Individual
TakingDamage	Bool	Individual
UnderFire	Bool	Individual
NextCoverLocation	Vector	Individual
NextAttackLocation	Vector	Individual
AttackMap	Map<String,Vector>	Global
SuppressionMap	Map<String,Vector>	Global
CurrentLocationIsSafe	Bool	Individual
NextCoverIsSafe	Bool	Individual
CurrentLocationIsAttack	Bool	Individual
NextLocationIsAttack	Bool	Individual
PlayerReference	Ref	Individual
PlayerIsVisible	Bool	Individual
IAmVisible	Bool	Individual
PlayerIsClose	Bool	Individual
PlayerIsLost	Bool	Individual

Tabla 8: Memoria del árbol de comportamiento Fight

5.2.5.5 *Funcionamiento*

Como ya comentamos anteriormente, este es el comportamiento más importante, y por ello también el más complejo (ver Figura 5.5). Éste ejecuta de manera contínua cada 0.25s el servicio *Shoot* (ver apartado 5.4.1.2) y se divide en tres ramas: el comportamiento de ocultación (ver apartado 5.2.5.5.1), el de avance (apartado 5.2.5.5.2) y el de ataque (apartado 5.2.5.5.3).

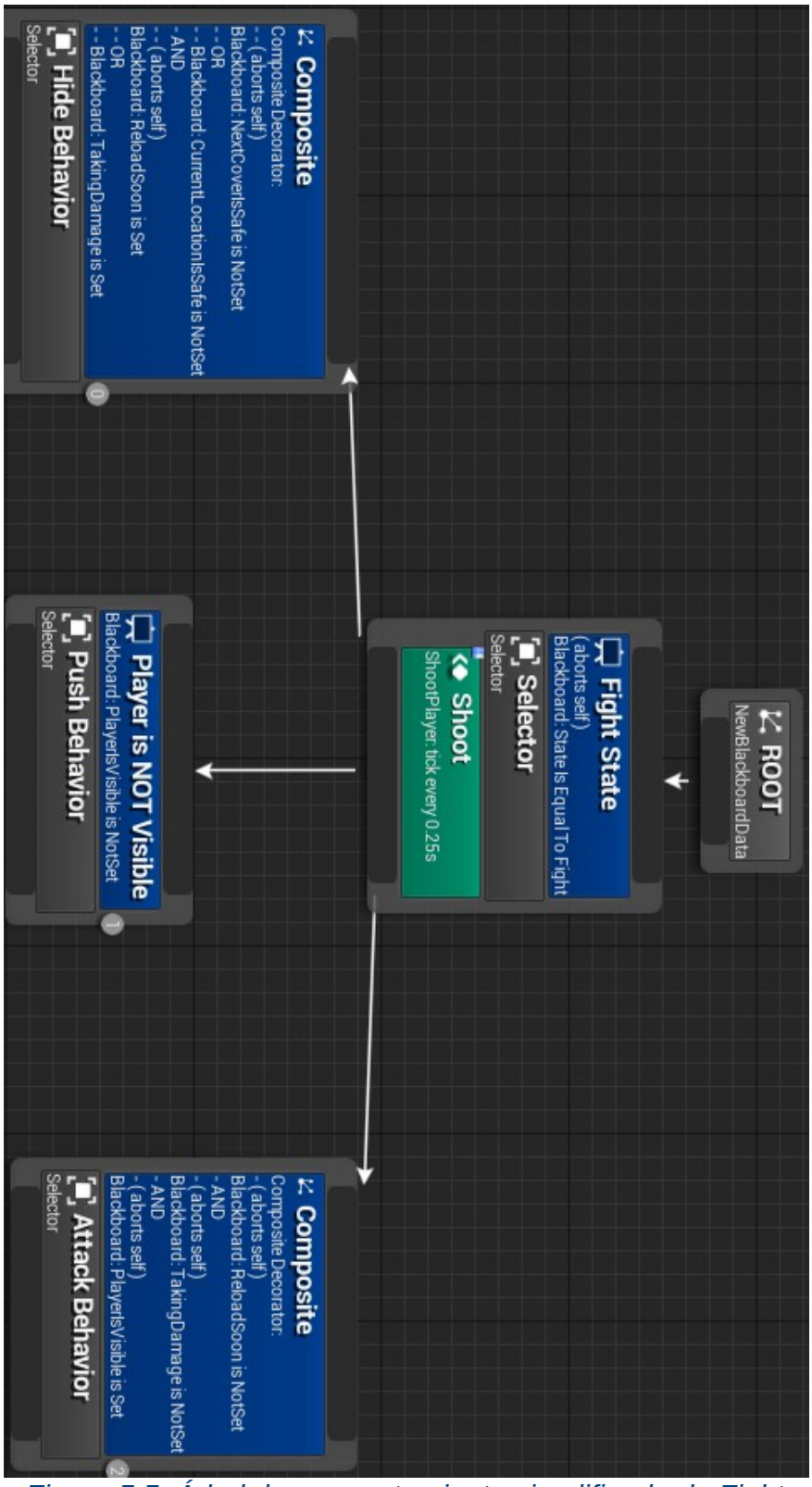


Figura 5.5: Árbol de comportamiento simplificado de Fight

5.2.5.5.1 Hide Behavior

5.2.5.5.1.1.1 Responsabilidad

Como su mismo nombre indica, la responsabilidad de este comportamiento es desplazar al agente a zonas seguras para ponerlo a salvo del enemigo.

5.2.5.5.1.1.2 Activación

La manera más fácil de explicar cuándo se activa este comportamiento es utilizar una expresión lógica (ver Figura 5.6).

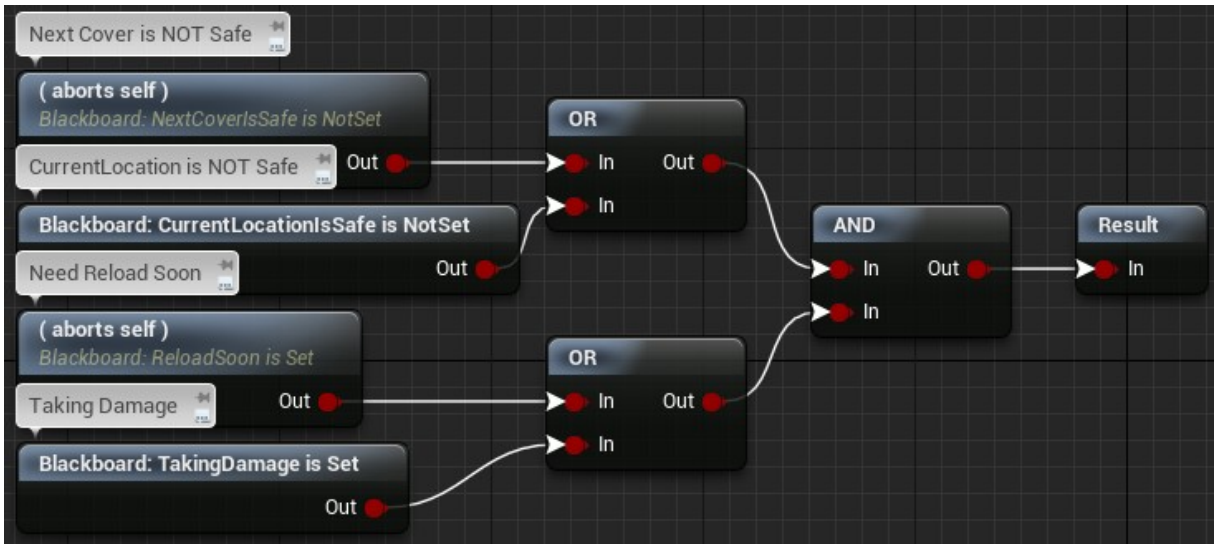


Figura 5.6: Condición de activación del comportamiento Hide

Esta expresión se podría traducir como:

(***!****CurrentLocationIsSafe* **||** **!**NextCoverIsSafe) **&&** (*ReloadSoon* **OR** *TakingDamage*)

Y se lee de la siguiente manera: si el personaje no esta a salvo (***!*** *CurrentLocationIsSafe*) o la cobertura a la que se dirige tampoco esta a salvo (**!** NextCoverIsSafe) y necesita recargar (*ReloadSoon*) o esta sufriendo daños (*TakingDamage),* en ese caso se evalúa a *True*.

5.2.5.5.1.1.3 Desactivación

Aunque este comportamiento requiere diversas variables para activarse, se desactiva tan pronto como el jugador recargue (**!***ReloadSoon)* o la cobertura a la que se dirige deje de ser segura (**!***NextCoverIsSafe).* Esta última situación puede surgir cuando el agente se desplaza a una posición que en principio parecía segura, pero que luego deja de serlo (por ejemplo porque el enemigo se ha movido). Por lo tanto, es necesario re-evaluar la situación y buscar una nueva cobertura.

También puede desactivarse el comportamiento si el padre de esta rama se desactiva. Es decir, si el agente deja de estar en el estado *Fight* se aborta la ejecución de toda la rama.

5.2.5.5.1.1.4 *Funcionamiento*

Su comportamiento es simple y parecido a los vistos anteriormente (ver Figura 5.7). Se trata de una secuencia de dos acciones:

0. Calcula y obtiene una localización segura y la asigna al siguiente punto de cobertura *NextCoverLocation* (ver apartado 5.4.2.4.3).
1. Se desplaza a la siguiente cobertura *NextCoverLocation por el camino más seguro* (ver apartado 5.4.2.3).

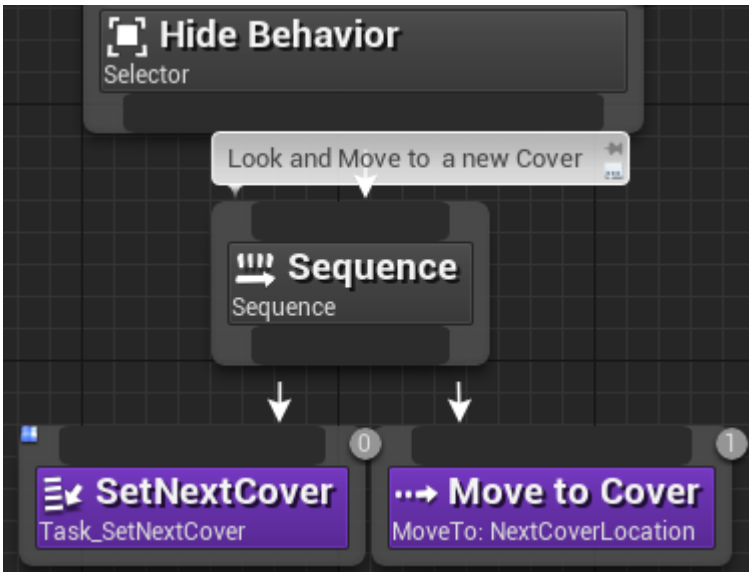


Figura 5.7: Rama de comportamiento correspondiente a la cobertura

5.2.5.5.2 *Push Behavior*

5.2.5.5.2.1.1 *Responsabilidad*

El objetivo de este comportamiento es aprovechar que el jugador se ha escondido para preparar el siguiente ataque y acercanos a él lo máximo posible para presionarlo. Además, mientras los agentes se desplazan también, aplican fuego de supresión para mantener al jugador acorralado.

5.2.5.5.2.1.2 *Activación*

En este caso la condición de activación es mucho más simple que antes (ver Figura 5.8). Simplemente es necesario que el jugador no sea visible (**!***PlayerIsVisible)* y que los comportamientos anteriores no se hayan activado (recordemos que los *BT* realizan una evaluación de izquierda a derecha, de más a menos prioritario).

Figura 5.8: Condición de activación del comportamiento Push

5.2.5.5.2.1.3 Desactivación

Una vez activo, sólo acaba su ejecución cuando ejecuta todas sus operaciones o cuando el padre es desactivado (cambio en la variable *State* del agente).

5.2.5.5.2.1.4 Funcionamiento

El funcionamiento de este árbol esta formado por dos partes: preparación y avance (ver Figura 5.9).

En la preparación, el agente aprovecha la situación para recargar el arma (ver apartado 5.4.2.1) si es necesario (*NeedReloadNow* **||** *NeedReloadSoon).*

Durante la rama de avance, si el agente tiene certeza de dónde se encuentra el jugador y no le ha perdido la pista, aplica fuego de supresión de manera contínua (ver apartado 5.4.1.3) y ejecuta una secuencia de dos acciones:

1. Calcula y obtiene una posición de ataque avanzada y la asigna a la variable *NextAttackLocation* (ver apartado 5.4.2.4.5). Para realizar este cálculo se utiliza la variable *AttackMap.*
2. Se desplaza al siguiente punto de ataque *NextAttackLocation* por el camino más seguro (ver apartado 5.4.2.3).

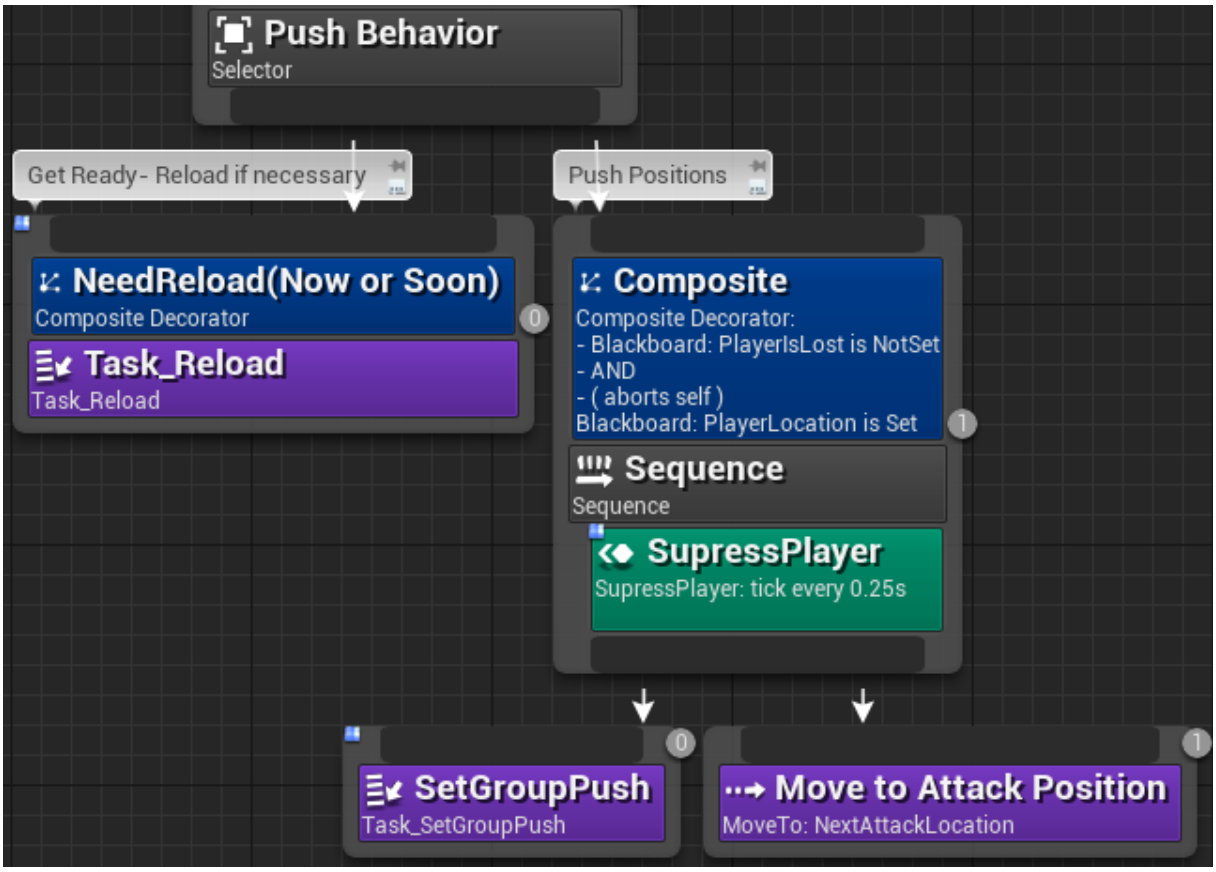


Figura 5.9: Rama de comportamiento correspondiente al avance

5.2.5.5.3 Attack Behavior

5.2.5.5.3.1.1 Responsabilidad

El objetivo de este comportamiento es posicionar al agente en una buena localización para atacar al jugador.

5.2.5.5.3.1.2 Activación

Este comportamiento se activa siempre que no se haya activado un comportamiento más prioritario y cuando el agente esta preparado para atacar (**!***NeedReloadSoon),* no ha sufrido daños recientemente (**!***TakingDamage)* y ve al jugador (*PlayerIsVisible)* (ver Figura 5.10).

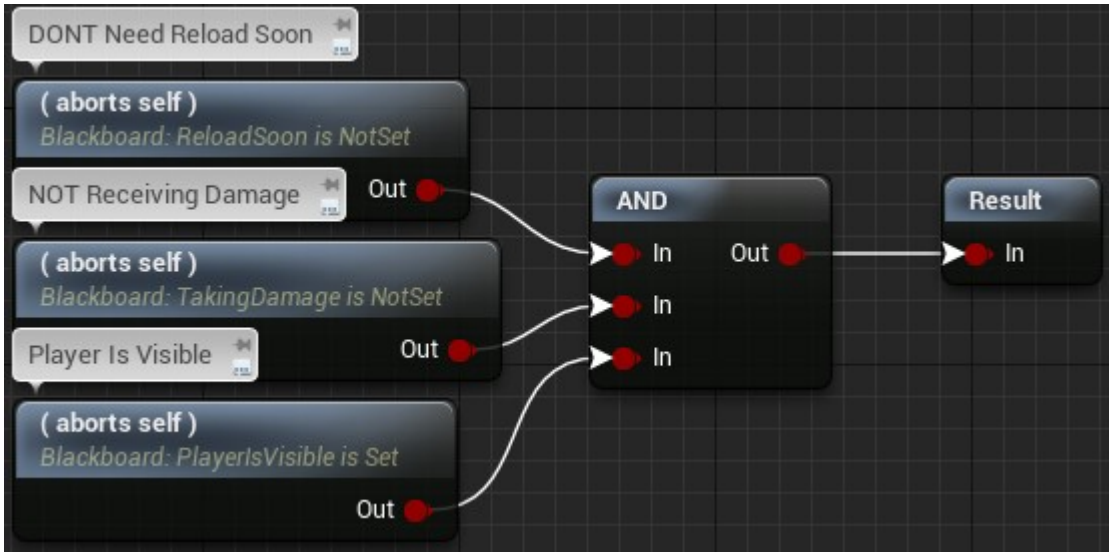


Figura 5.10: Condición de activación del comportamiento Attack

5.2.5.5.3.1.3 *Desactivación*

En el mismo instante que el agente necesite recargar (*NeedReloadSoon),* o reciba una cantidad considerable de daño (*TakingDamage)* o deje de ver al jugador (**!** PlayerIsVisible), se desactiva este comportamiento.

Al igual que antes, si el padre de esta rama se desactiva, este comportamiento también se desactiva.

5.2.5.5.3.1.4 *Funcionamiento*

Este comportamiento se divide en dos ramas, similares pero con objetivos diferentes (ver Figura 5.11)

La primera rama se activa cuando el enemigo y el agente están muy cerca (ver más detalles en el apartado 5.3.3.2) y simplemente el agente se mueve directo hacia su posición utilizando el camino más corto.

En la otra rama, si el jugador no esta cerca (*!PlayerIsClose)* se ejecuta un selector con tres operaciones:

1. Si (*CurrentLocationIsAttack* **&&** **!***UnderFire)* entonces *Mantener posición* (ver apartado 5.4.2.2)
2. Si (*NextLocationIsAttack* **&&** **!***UnderFire)* entonces *Desplazarse a la siguiente posición de ataque por el camino más seguro* (ver apartado 5.4.2.3)
3. Si (*UnderFire* **||** (**!***NextLocationIsAttack* **&&** **!***CurrentLocationIsAttack*)) entonces ejecutará una secuencia de dos acciones:

   A) Calcula y obtiene una nueva posición de ataque y la asigna a la variable *NextAttackLocation*(ver apartado 5.4.2.4.4). Para realizar este cálculo se utiliza la variable *AttackMap.*

B) Se desplaza a la localización *NextAttackLocation* por el camino más seguro (ver 5.4.2.3)

Es importante destacar que, si mientras el agente está manteniendo la posición de ataque, su posición deja de ser buena (**!***CurrentLocationIsAttack),* entonces la rama se desactiva automáticamente.

De la misma manera, si el jugador se esta moviendo hacia su última posición de ataque y esta deja de ser buena (**!***NextLocationIsAttack),* también se desactivará.

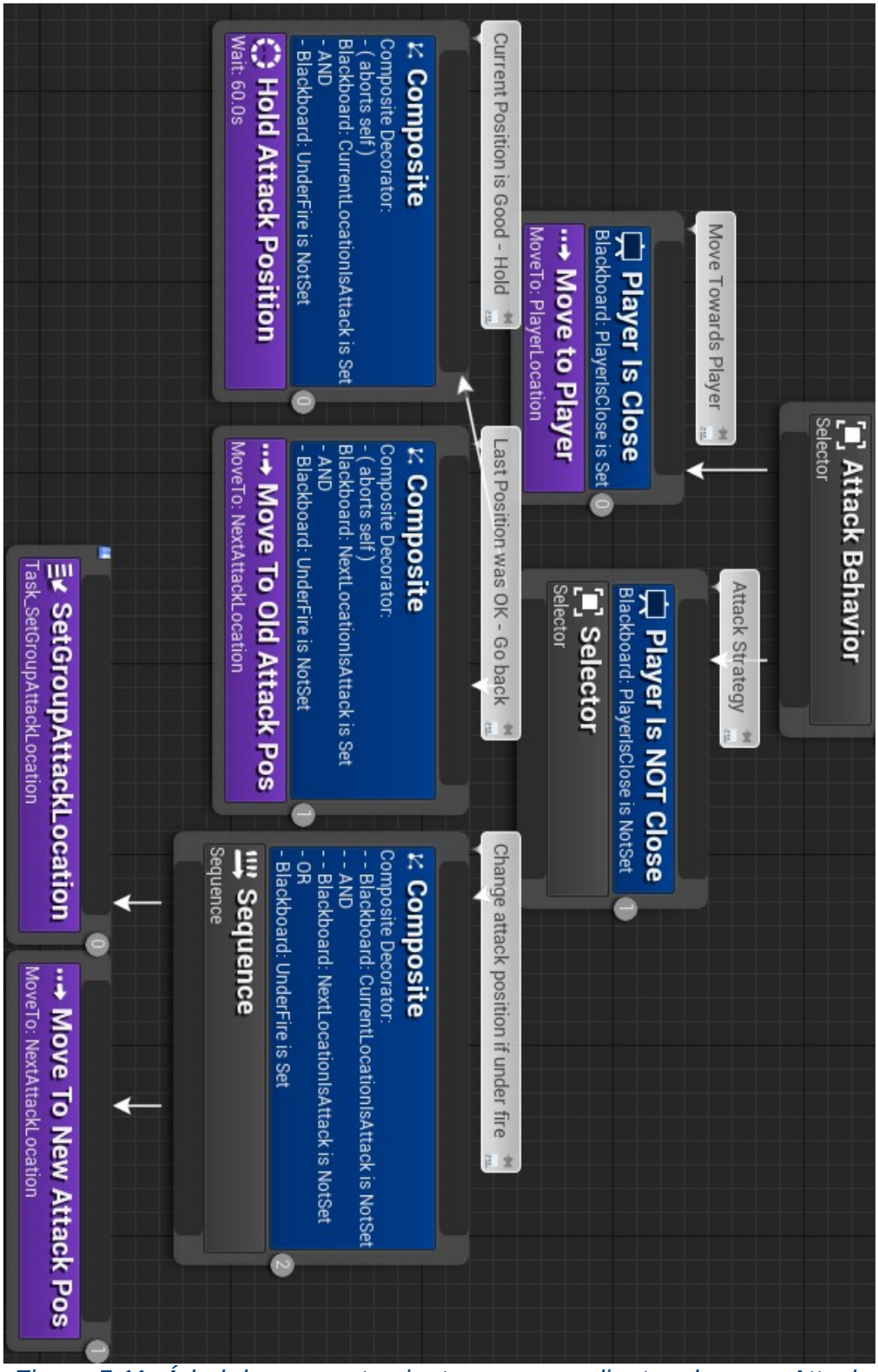


Figura 5.11: Árbol de comportamiento correspondiente a la rama Attack

5.3 Sentir

Una vez explicado el razonamiento del agente, el siguiente paso para entender comportamiento es explicar como se obtiene toda la información que el sistema anterior utiliza para tomar las decisiones.

Esta información se puede clasificar en tres categorías dependiendo de la fuente de origen de los datos: percepción, introspección y cálculo.

Mientras que la percepción es *event-driven* y se ejecuta únicamente cuando se percibe al usuario, la información de introspección y la de cálculo se actualizan aproximadamente a una frecuencia de 3.33Hz (ver Figura 5.1).

5.3.1 *Percepción*

La información obtenida por la percepción hace referencia únicamente al jugador (*PlayerRef)* y a su posición (*PlayerLocation*). Este sistema se puede dividir a su vez en el sentido de la vista y del oído. Estos sentidos permiten al *NPC* ver y oír al jugador con el fin determinar su posición con exactitud.

El funcionamiento de todos los sistemas de percepción de *Unreal*, independientemente del sentido, es el mismo. Cuando el sistema percibe un cambio (un sonido, un enemigo, daño, etc) se activa y se ejecuta una función específica con la información del instigador que ha generado el cambio.

5.3.1.1 Vista

El sistema de percepción encargado de ver se configura mediante dos parámetros:

- **Angulo de visión.** Determina el ángulo con el que ve el *NPC.* Aunque para los humanos está alrededor de 180º, en los juegos *FPS* el ángulo de la cámara no suele llegar a los 60º.
- **Rango.** Distancia máxima a la que puede ver.

Su funcionamiento es muy sencillo. En cuanto el jugador entra o sale del rango y ángulo de visión del *NPC,* se activa un evento que llama a una función especifica.

Esta función es la encargada de asignar a la variable *PlayerRef* la referencia del jugador si esté entra en la zona y rango de visión, o asignarle un valor nulo si el jugador sale.

De esta manera, si el *NPC* ve al jugador, su referencia está guardada en la variable *PlayerRef* y si no, esta variable tiene un valor nulo. Más adelante veremos como otra función determina de manera periódica la posición del jugador a partir de la variable *PlayerRef* (ver apartado 5.3.3.3).

5.3.1.2 Oído

Por lo que respecta al sistema de percepción del oído sólo está determinado por un

parámetro: el rango, que al igual que antes define la distancia máxima a la que puede oír.

Su funcionamiento es muy similar al de la vista. En cuanto el jugador emite un ruido (concretamente cuando dispara) dentro del rango de escucha se activa un evento.

Este evento llama a una función que, a diferencia de antes, no modifica la variable *PlayerRef* ya que no seria justo para el jugador. Únicamente modifica la variable *PlayerLocation* asignándole la posición desde la cual se ha emitido el sonido.

5.3.2 Introspección

Cada agente inteligente es capaz de conocer datos relativos a él mismo, como su posición, munición actual, orientación, etc. A continuación explicaremos los diferentes datos que el agente analiza sobre sí mismo para tomar decisiones.

5.3.2.1 Estado

La variable *State* se modifica de acuerdo a un conjunto de condiciones y se puede expresar como una máquina de estados (ver Figura 5.12).

Comenzamos en el estado de patrulla (*Patrol).* Permanecemos en este estado hasta que escuchamos o vemos al jugador (mediante el sistema de percepción) en algún momento entonces cambiamos al estado de búsqueda (*Search)* si lo hemos oído o al estado de combate (*Fight)* si lo vemos.

Entonces, desde el estado *Search* únicamente pasamos al estado *Fight* si vemos al jugador. Del estado *Fight,* si por algún motivo le perdemos la pista al jugador (*PlayerIsLost)*, volvemos al estado de búsqueda *Search.* Por último, si el jugador muere, pasamos de manera permanente al estado *Idle.*

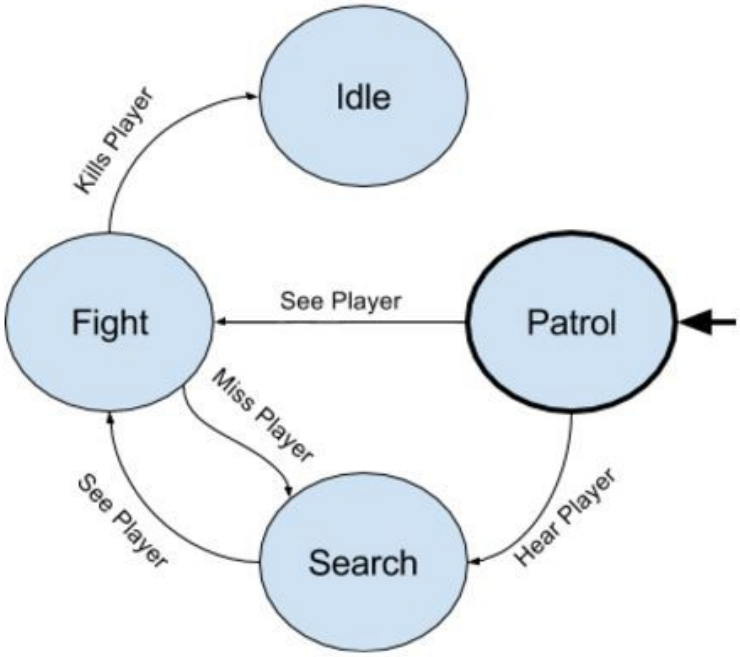


Figura 5.12: Diagrama de estados describiendo el comportamiento de la variable State

5.3.2.2 *Munición*

Modifica las variables relativas a la munición del jugador y que le indican cuándo tiene que recargar:

- *NeedReloadSoon.* Se activa cuando el porcentaje de munición en el cargador actual es inferior al 40%.
- *NeedReloadNow.* Se activa cuando el porcentaje de munición en el cargador actual es inferior al 10%.

5.3.2.3 *Vida*

Para que el agente sea capaz de percibir los daños que le hacen y actuar de manera consecuente, existe la variable *TakingDamage.* Esta variable esta activada si el agente ha recibido en los últimos segundos un daño igual o mayor a 15 puntos de vida.

5.3.2.4 *Combate*

Para saber si el agente se encuentra en combate utilizamos la variable *UnderFire.* Esta variable se activa inmediatamente después que el agente haya recibido algún tipo de daño y permanece activa durante 7 segundos. Idealmente esta variable

debería activarse cuando estamos bajo fuego enemigo. Es decir, si alguien nos dispara, acierte o no. No obstante, dada la dificultad de este problema, he decidido utilizar esta aproximación simple que funciona satisfactoriamente.

5.3.3 Cálculo

Mediante el uso de operaciones matemáticas podemos realizar cálculos para obtener información extra sobre el agente, el entorno y el jugador enemigo.

Cabe destacar que muchos de los cálculos que explicaremos a continuación están estrechamente relacionados con el jugador. Por lo tanto, para realizar los cálculos utilizaremos la variable *PlayerRef.* En los casos en los cuales esta variable no sea válida, tendremos que realizar algún tipo de predicción o estimación.

5.3.3.1 Visibilidad del jugador

Esta función tiene la responsabilidad de determinar el área visible por jugador y si el agente es visible (*IAmVisible).* Obviamente, para realizar estos cálculos necesitamos la referencia al jugador *PlayerRef.*

Si tenemos esta referencia, simplemente calculamos su visibilidad (ver más detalles en el apartado 6.2) y modificamos los costes de la *NavMesh* con el fin de que los agentes interpreten como más costosas las zonas visibles por el jugador (ver apartado 6.3).

Para calcular si el agente es visible (*IAmVisible) podríamos* comprobar si se encuentra dentro de esta zona visible por el jugador, pero es más rápido hacerlo de otra manera. Si calculamos el ángulo entre el vector de dirección del jugador con el vector que une al jugador y al agente (ver Figura 5.13), y si este ángulo esta dentro del angulo de visión del jugador, *IAmVisible* será cierto.

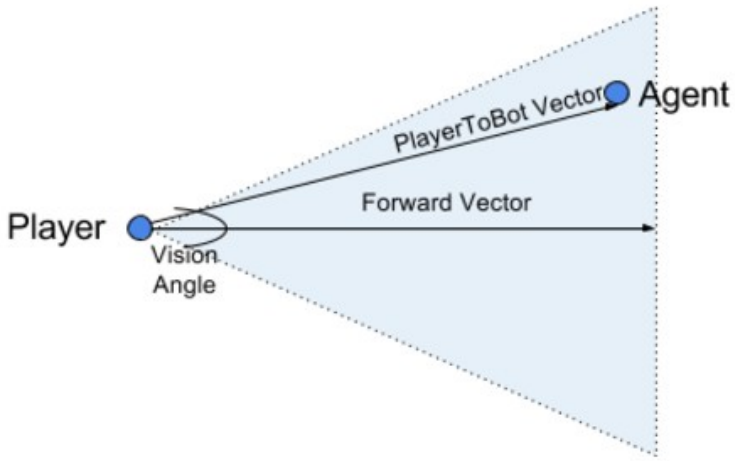


Figura 5.13: Calculo de la visibilidad del agente respecto del jugador

En el caso en que no tengamos la referencia al jugador, no realizaremos ningún tipo de predicción. Simplemente diremos que no somos visibles (*IAmVisible* sera falso) y la zona visible por el jugador será nula, por lo que la *Navmesh* utilizará únicamente una heurística basada en distancia euclidiana.

5.3.3.2 Distancia al jugador

El objetivo de esta función es determinar si el jugador está muy cerca del agente y asignar la variable *PlayerIsClose* en consecuencia. Para ello, necesitamos la referencia al jugador *PlayeRef* para ser capaces de medir la distancia que separa la posición del jugador y la posición del agente. Si esta distancia es más pequeña que un cierto *threshold. PlayerIsClose* será cierto.

Si la distancia es más grande que el *threshold* o no tenemos una referencia válida al jugador, *PlayerIsClose* será falso.

5.3.3.3 Localización del jugador

La localización del jugador *PlayerLocation* se puede obtener de manera directa a partir de la referencia del jugador *PlayerRef.*

No obstante, si esta referencia no es válida, utilizaremos la última posición del jugador como aproximación y ejecutaremos un sistema de predicción basado en mapas de influencia para determinar la posición más probable del jugador (ver apartado 6.4). De esta manera, si el jugador se ha movido de su ultima localización, el mapa de influencia nos ayudara a estimar a dónde habrá ido.

5.3.3.4 Jugador en paradero desconocido

El agente determina que ha perdido, o que desconoce, la posición del jugador (*PlayerLost)* siempre y cuando el agente sea capaz de ver la última localización del jugador (*PlayerLocation)* y no vea al jugador.

Por lo tanto, la activación de esta variable indica una necesidad de cambiar al estado *Search* para encontrar al jugador.

5.3.3.5 Actualización de posiciones tácticas

La actualización de posiciones tácticas consiste en determinar la **calidad táctica** de un conjunto de posiciones para cubrirse o para atacar al jugador.

En concreto, el agente inteligente evalúa, por un lado, la calidad táctica de defensa de la cobertura actual *CurrentLocationIsSafe* y de la siguiente cobertura *NextCoverIsSafe.* Y por otro lado la calidad táctica de ataque de la posición actual y la siguiente, *CurrentLocationIsAttack* y *NextLocationIsAttack,* respectivamente.

Dependiendo del tipo de posición, la calidad táctica está determinada por factores diferentes (ver 5.4.2.4).

5.4 Actuar

Por último, falta explicar el funcionamiento específico de las diferentes acciones que ejecutan los distintos comportamientos.

Estas acciones en *UE4* corresponden siempre a servicios o a tareas (nodos hoja) y su objetivo siempre es el mismo: modificar el estado del agente o del entorno.

5.4.1 Servicios

Los servicios, como ya cometamos anteriormente (ver apartado 4.2.3), son funciones que se ejecutan de manera periódica mientras el nodo está activo.

A continuación explicaremos de manera más detallada el funcionamiento de los servicios que ya hemos mencionado previamente durante la explicación del comportamiento: Detección y actualización de datos, Disparar y Supresión.

5.4.1.1 *Detección y actualización de datos*

Este servicio es el encargado de ejecutar todas las funciones responsables de captar la información del entorno (ver apartado 5.3) a excepción de la información que se capta mediante el sistema de percepción (ver apartado 5.3.1) ya que es *event-driven.* Este servicio se ejecuta aproximadamente a una frecuencia de 3.33Hz (ver recuadro verde en la Figura 5.1).

5.4.1.2 *Disparar*

El servicio de disparar se ejecuta de manera continua mientras el agente se encuentra en el estado *Fight* (ver recuadro verde en la Figura 5.5).

Este servicio es muy simple, si el jugador es visible *PlayerIsVisible* y tiene una linea clara de tiro (no hay otros agentes entre su linea de tiro y el jugador) dispara. Si la linea de tiro no es clara, el agente busca una posición cercana (izquierda o derecha) con una linea de tiro clara antes de disparar. Por último, si se queda sin balas, recarga automáticamente (ver la implementación detallada en el apartado 6.5).

5.4.1.3 *Supresión*

El servicio de supresión sólo se ejecuta en la rama de avance durante una situación de combate (ver Figura 5.9). Su objetivo es acorralar, presionar el jugador para que se mantenga escondido mientras los agentes avanzan. Los agentes, de alguna manera, tienen que predecir los siguientes movimientos del jugador. Una vez han calculado estas posiciones, comprueban que no hay agentes en su linea de tiro y disparan para aplica fuego de supresión, evitando así que el jugador pueda moverse o atacar (leer más en el apartado 5.4.2.4.6). La implementación detallada se encuentra en el apartado 6.5.

5.4.2 Tareas

Las tareas o nodos hoja, a diferencia de los servicios, son nodos que solamente se ejecutan una única vez cuando están activos y automáticamente devuelven si se ejecutaron con éxito o no.

5.4.2.1 Reload

Tal y como su nombre indica esta tarea indica al agente que tiene que recargar el arma. El agente puede haber gastado sólo una bala, diez o todas. Si recibe la orden de recargar, recarga. Cuando acaba de recargar la tarea acaba.

5.4.2.2 Wait

Esta es probablemente la tarea más simple pero no por eso la menos útil. Permite ejecutar una tarea que no hace nada y que tiene una duración especifica a través de dos parámetros:

- **Tiempo de espera.** El tiempo base que durara la tarea.
- **Desviación.** La desviación positiva o negativa que queremos aplicar al tiempo de espera base.

Esta tarea nos permite simular, entre otras cosas, el pensamiento de los agentes. Esto nos permite generar una sensación de que el agente esta razonando y que sus decisiones no son inmediatas. Ademas, poder especificar un parámetro desviación nos permite agregar cierta aleatoriedad y diversidad al sistema para que resulte menos predecible.

5.4.2.3 Move To

Esta tarea permite al agente moverse de su posición actual a una nueva posición que le especifiquemos. La tarea finaliza satisfactoriamente cuando el agente llega a su destino.

Para encontrar el mejor camino el agente ejecuta una operación de *pathfinding* sobre la *Navmesh.* La definición de “mejor camino” depende del comportamiento y de la situación del agente y su entorno. Por esta razón, dependiendo de la situación, los agentes escogen entre dos definiciones de “mejor camino”: según distancia o seguridad.

5.4.2.3.1 Mejor camino: distancia

La mayoría de ocasiones nos interesa encontrar el camino más corto entre dos puntos. Para ello podemos utilizar el algoritmo de *pathfinding* de *Unreal* (ver apartado 4.2.2) que utiliza un algoritmo *A** con una heurística basada en distancia euclidiana (ver Figura 5.14).

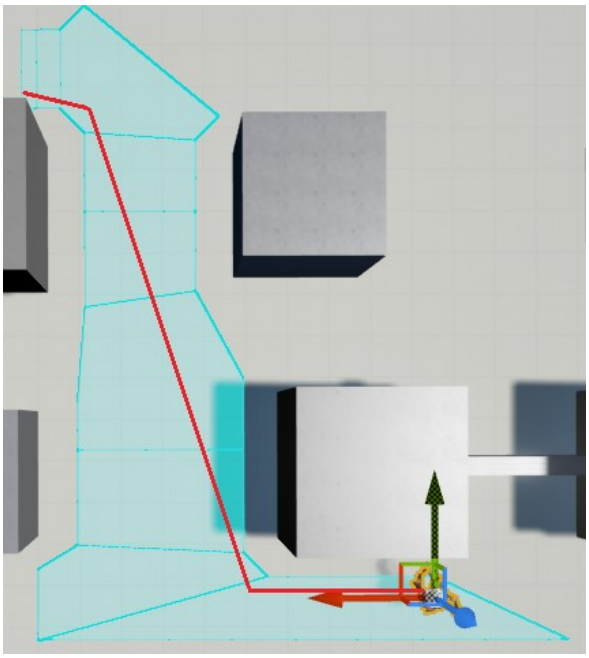


Figura 5.14: Algoritmo A con una heurística basada en distancia euclidiana. Polígonos evaluados y camino elegido*

5.4.2.3.2 *Mejor camino: seguridad*

En situaciones de combate, la importancia que tiene la distancia sobre un camino es mucho menor que la que puede tener su seguridad. Lo que nos interesa es encontrar el camino más seguro que una dos puntos, evitando zonas conflictivas o peligrosas. En nuestro caso, las zonas conflictivas o peligrosas, son aquellas que el jugador ve y por tanto en las que nos puede disparar. Recordemos que para marcar estas zonas como peligrosas en la *Navmesh* primero tenemos que calcular la visibilidad del jugador (ver apartado 5.3.3.1) y después modificar la *Navmesh* (ver apartado 6.3).

Por lo tanto, el objetivo es encontrar un camino lo más seguro y corto posible.

5.4.2.4 *Cálculos tácticos*

Muchas de las tareas que hemos visto durante el análisis de los árboles de comportamiento corresponden al cálculo de posiciones tácticas (búsqueda, cobertura, ataque...). Para los seres humanos, determinar si una posición es buena para cubrirse o atacar es muy fácil e intuitivo. Lo cierto es que sin darnos cuenta, nuestro cerebro tiene en cuenta muchas variables a la hora de discriminar entre las diferentes posiciones.

Para este proyecto, hemos profundizado en este razonamiento y hemos replicado los que creemos que son los factores más importantes a tener en cuenta a la hora de evaluar tácticamente una posición.

Gracias al sistema *EQS* de *Unreal*, convertir estos factores a condiciones o tests a la hora de evaluar el entorno no ha sido muy complicado (ver apartado 6.6).

A continuación detallaremos en qué consisten cada una de las siguientes consultas tácticas, así como los tests que utilizan para determinar cuál es la mejor posición.

5.4.2.4.1 Punto de patrulla

La verdad es que el cálculo de puntos de patrulla no es para nada táctico. Simplemente itera una lista de puntos de patrulla predefinida y devuelve el punto correspondiente al índice actual, más uno (ver Figura 5.15).

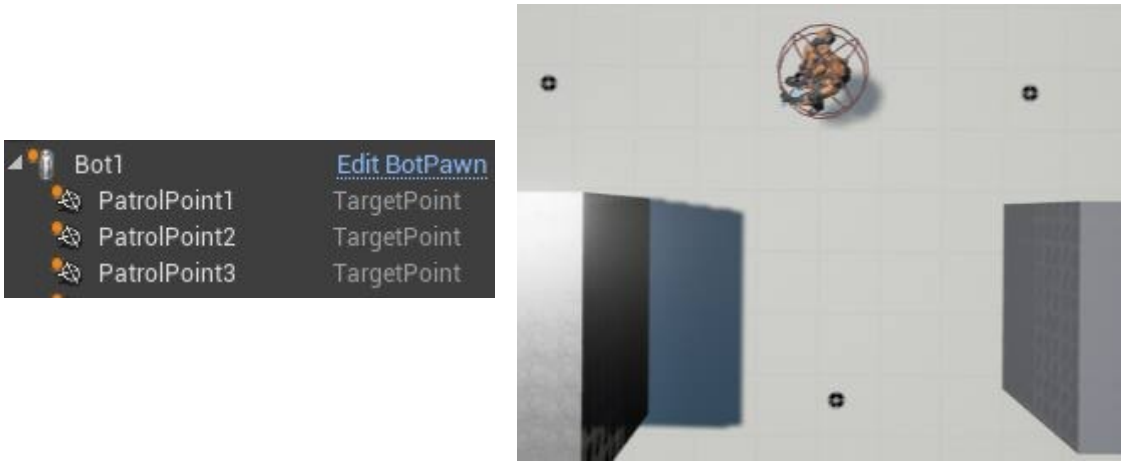


Figura 5.15: Puntos de patrulla de un NPC. A la izquierda la lista de puntos y a la derecha los puntos situados en el espacio

5.4.2.4.2 Punto de búsqueda

Para calcular tácticamente el mejor punto de búsqueda debemos tener en cuenta varios factores:

1. Las posiciones a las que puede haber llegado el jugador desde la última vez que lo vimos.
2. Preferir posiciones cercanas a nosotros para minimizar el tiempo de desplazamiento.
3. Preferir posiciones lejanas a otros agentes para cubrir más terreno.
4. El punto esté cerca de una posición de cobertura para escondernos si la situación empeora.

Definidos estos cuatro puntos podemos traducirlos fácilmente en cuatro tests dentro de una consulta *EQS.* De más a menos importantes estos tests son:

1. Test de influencia (ver apartado 6.6.2.1). Utilizaremos este test para determinar en qué posiciones podría estar el jugador con una alta probabilidad (alta influencia).

2. Test de coste (ver apartado 6.6.1.1). Con este test favoreceremos aquellos puntos que tengan un coste bajo según el algoritmo de *pathfinding.* En este caso un coste bajo se corresponde con caminos cortos.
3. Test de expansión (ver apartado 6.6.2.2). Favoreceremos aquellas posiciones que estén lo más alejadas posibles de las posiciones de búsqueda de los otros agentes (variable *SearchMap)*. De esta manera cubriremos una mayor cantidad de terreno
4. Test de proximidad a los obstáculos (ver apartado 6.6.1.5). Puntuaremos de manera más alta aquellas posiciones cercanas a un obstáculo. De esta manera, favoreceremos que el agente permanezca cerca de posiciones potenciales de cobertura.

Si juntamos todos estos tests y construimos la consulta *EQS* obtenemos aquellas posiciones en las que probablemente esté el jugador, que están más cerca nuestro y lejos de los otros agentes y ademas que están cerca de una cobertura potencia. En la Figura 5.16 dos agentes están buscando al jugador escondido detrás de un obstáculo. Podemos ver más detalles y una demostración de su funcionamiento en el apartado 8.5.

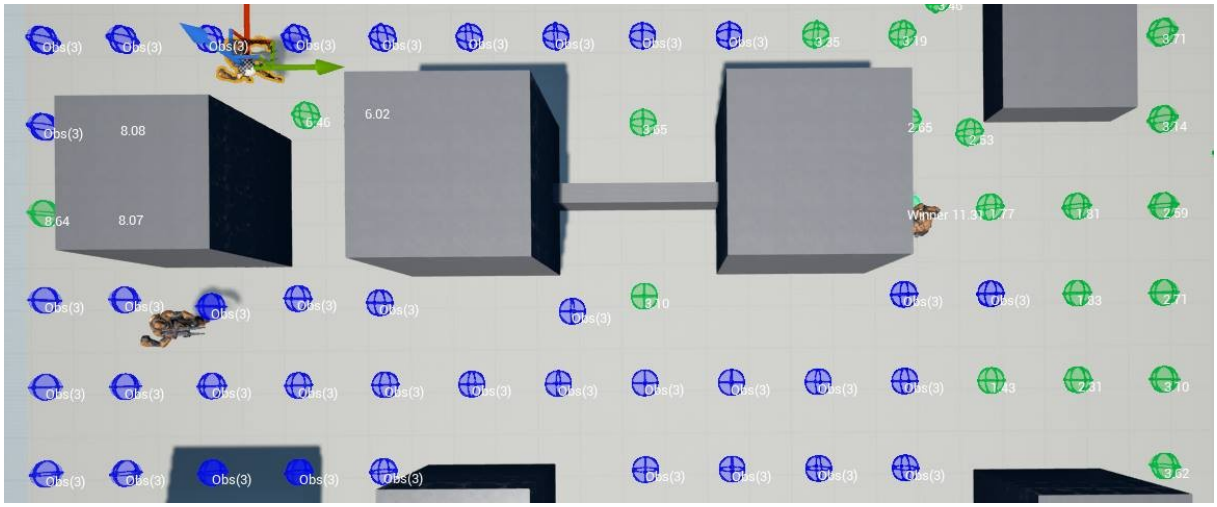


Figura 5.16: Resultado de la consulta EQS de búsqueda

5.4.2.4.3 Punto de cobertura

Para calcular el mejor punto de cobertura que nos permita escondernos de una manera y rápida y segura del jugador, tenemos que analizar:

- Que la posición este cerca de un obstáculo y que este nos proporcione una zona que no sea visible por el jugador en la que podamos escondernos.
- Que esté cerca.

Si transformamos esos requisitos en una consulta EQS con dos tests, de más a menos importantes, obtenemos:

1. Test “detrás” de los obstáculos (ver apartado 6.6.1.4). Eliminaremos todas aquellas posiciones que no estén detrás de un obstáculo teniendo en cuenta la posición del jugador. Ademas, las puntuaremos de manera inversamente proporcional respecto la distancia de la posición al obstáculo en la dirección del jugador.
2. Test de coste (ver apartado 6.6.1.1). Con este test favoreceremos aquellos puntos que tengan un coste bajo según el algoritmo de *pathfinding.* En este caso, el coste se calculará teniendo en cuenta la seguridad y la distancia.
3. Test de aleatoriedad (ver apartado 6.6.1.6). Con este test simplemente agregamos un 5-10% de aleatoriedad en la puntuación de los resultados para que los resultados no sean siempre los mismos.

Si ejecutamos esta consulta *EQS* (ver Figura 5.17) obtenemos aquellas posiciones que están detrás de un obstáculo y por tanto, que son seguras. Ademas, las posiciones están puntuadas de acuerdo al coste de desplazarnos hacia ellas, teniendo en cuenta la seguridad y la distancia. Más detalles y demostración de su funcionamiento en el apartado 8.6.1.

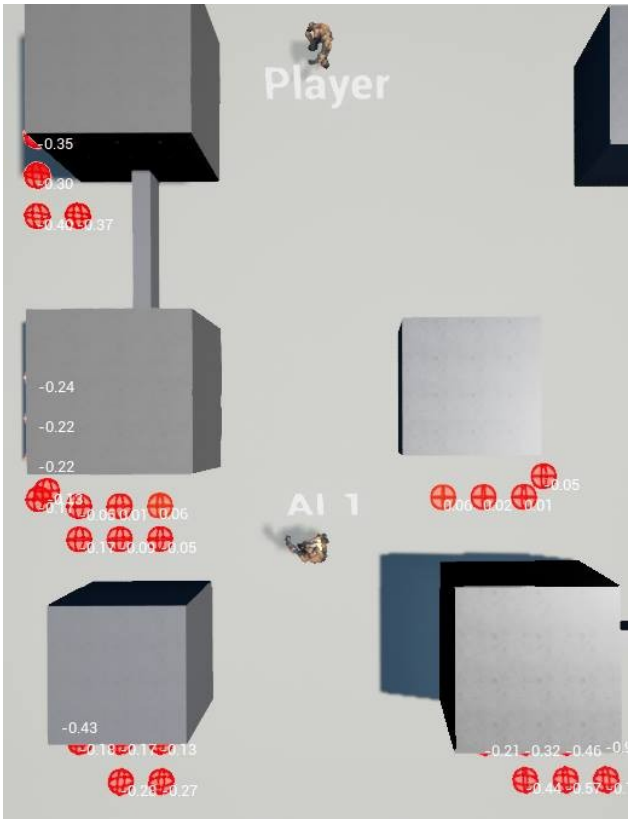


Figura 5.17: Resultado de la consulta EQS de cobertura

5.4.2.4.4 Punto de ataque

Por lo que respecta a la posición de ataque es algo más compleja que las anteriores por lo que debemos tener en cuenta muchos más factores:

- El jugador debe ser visible desde esta posición (tenemos que verlo para poder atacar).
- Debe esta cerca de una posición de cobertura (definida en el apartado anterior).
- Debe ser una posición lo más diferente posible de las ya elegidas por los otros agentes. De esta manera, los agentes se distribuirán para atacar al jugador desde diferentes flancos.
- Debe estar cerca del agente para evitar peligros.
- Debe estar cerca del jugador para así intentar avanzar a la misma vez que atacamos.

Si traducimos estos factores en tests dentro de una consulta EQS, el resultado, de más a menos importante, es:

1. Test de visibilidad (ver apartado 6.6.1.3). Sólo nos interesa evaluar aquellas posiciones desde las cuales se puede ver al jugador para así poder atacarlo. Por esta razón, eliminaremos todas las posiciones que no tengan visibilidad al jugador.
2. Test de proximidad a los obstáculos (ver apartado 6.6.1.5). Puntuaremos de manera más alta aquellas posiciones cercanas a un obstáculo. De esta manera, favoreceremos que el agente permanezca cerca de posiciones potenciales de cobertura por lo tanto que pueda esconderse rápidamente cuando lo necesite.
3. Test de flanqueo (ver apartado 6.6.2.3). Favoreceremos aquellas posiciones que sean lo más diferentes posibles de las posiciones de ataque ya escogidas (variable *AttackMap).* De esta manera, las posiciones de ataque se distribuirán creando una ofensiva desde diferentes flancos.
4. El objetivo de este test es favorecer aquellas posiciones que permitan rodear al jugador para atacarlo por diferentes flancos.
5. Test de coste (ver apartado 6.6.1.1). Con este test favoreceremos aquellos puntos que tengan un coste bajo según el algoritmo de *pathfinding.* En este caso, el coste se calculará teniendo en cuenta la seguridad y la distancia.
6. Test de distancia (ver apartado 6.6.1.2). Con este test favorecemos aquellas posiciones que estén más cerca del jugador. Es decir, que la distancia sea más baja. De esta manera intentamos que los agentes se acerquen al jugador.

7. Test de aleatoriedad (ver apartado 6.6.1.6). Con este test simplemente agregamos un 5-10% de aleatoriedad en la puntuación de los resultados para que los resultados no sean siempre los mismos.

En la Figura 5.18 vemos el resultado de ejecutar este consulta *EQS* de ataque. Todas las posiciones que vemos tienen visibilidad directa al jugador y están puntuadas de acuerdo a la distancia al agente que hace la consulta, al jugador y teniendo en cuenta las otras posiciones de ataque de los otros agentes (ver el apartado 8.6.3 para una demostración de su funcionamiento).

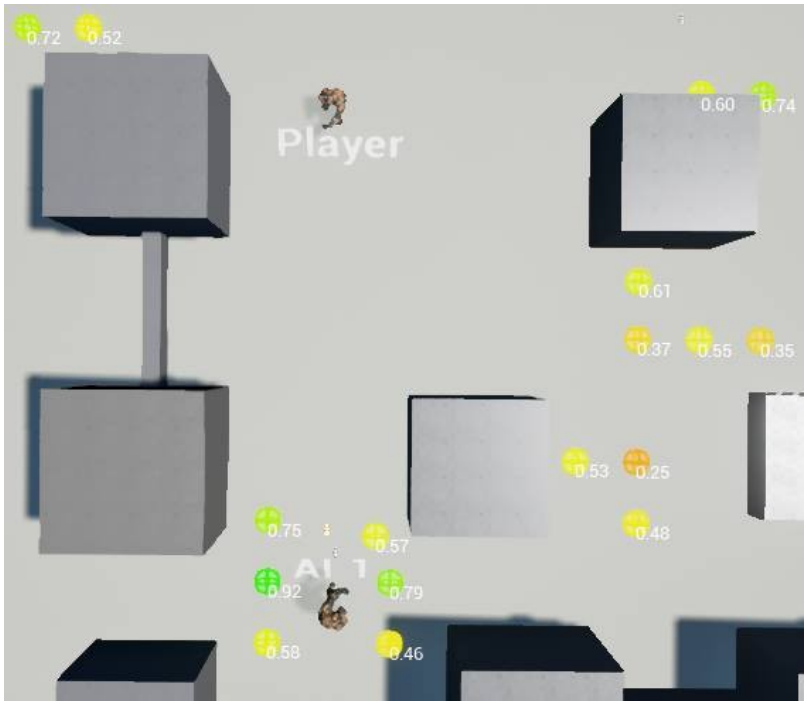


Figura 5.18: Resultado de la consulta EQS de ataque

5.4.2.4.5 Punto de avance

Para el cálculo del punto de avance, consideramos que es igual que el punto de ataque pero incrementando la importancia del test que evalúa distancia al jugador. De esta manera se favorecen más las posiciones próximas al jugador y conseguimos este avance al atacar (ver Figura 5.19). En comparación con la Figura 5.18 vemos que aquí las posiciones mejor puntuadas son las cercanas al jugador.

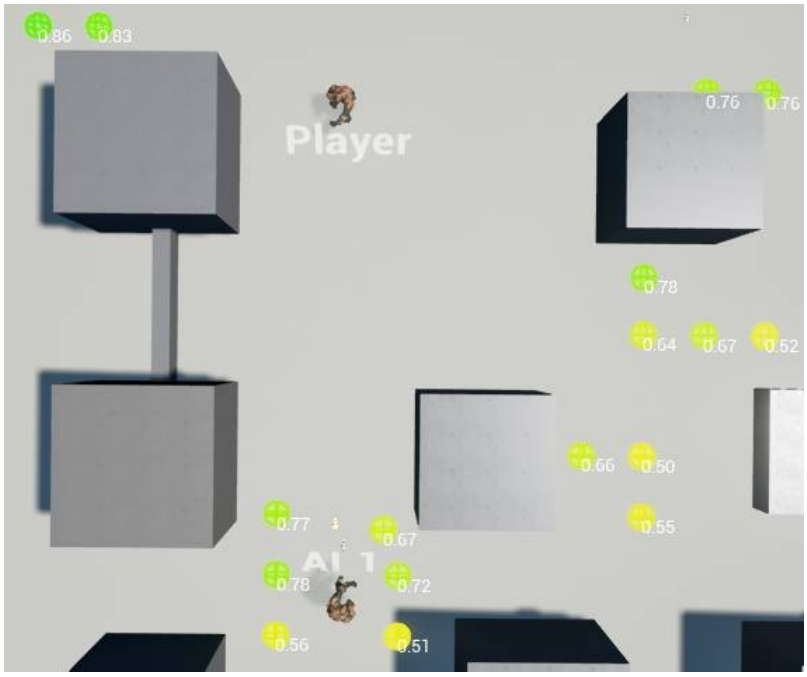


Figura 5.19: Resultado de la consulta EQS de avance

5.4.2.4.6 Punto de supresión

El cálculo del punto de supresión es diferente de todos los anteriores. Mientras los anteriores siempre giraban entorno al agente, el punto de supresión gira entorno al jugador. De alguna manera, el agente tiene que predecir la siguiente posición del jugador y disparar a ese punto de supresión. Lo más importante para calcular estos puntos es:

1. La visibilidad del agente a este punto.
2. La distancia del jugador al punto de supresión. Tiene que ser el primer punto de exposición del jugador.
3. Que los puntos de supresión de diferentes agentes sean diferentes. No tiene sentido que todos disparen al mismo punto de supresión. Deben repartirse o simplemente, que sólo dispare uno.

Si transformamos esos requisitos en una consulta EQS con cuatro tests:

1. Test de visibilidad (ver apartado 6.6.1.3). Solo nos interesa evaluar aquellas posiciones que son visibles para el agente y que puede atacar. Por esta razón, eliminaremos todas las posiciones a las que no tengamos visibilidad.
2. Test de distancia (ver apartado 6.6.1.2). Con este test favorecemos aquellas posiciones que estén más cerca del jugador. Es decir, que la distancia sea más baja.
3. Test de repetición (ver apartado 6.6.2.4). Con este test eliminamos todas las

posiciones cercanas a posiciones de supresión ya escogidas por otros agentes (variable *SuppressionMap)*. De esta manera evitamos que todos disparen a un mismo punto.

4. Test de aleatoriedad (ver apartado 6.6.1.6). Con este test simplemente agregamos un 5-10% de aleatoriedad en la puntuación de los resultados para que los resultados no sean siempre los mismos.

El resultado de esta consulta lo podemos ver en la Figura 5.20: Resultado de una consulta EQS para obtener los puntos de supresión donde las puntuaciones marcadas con 1 son aquellas elegidas por cada uno de los *NPCs* como posiciones de supresión. Estas posiciones son las a las que los *NPCs* tienen visibilidad y que ademas están muy cerca al jugador.

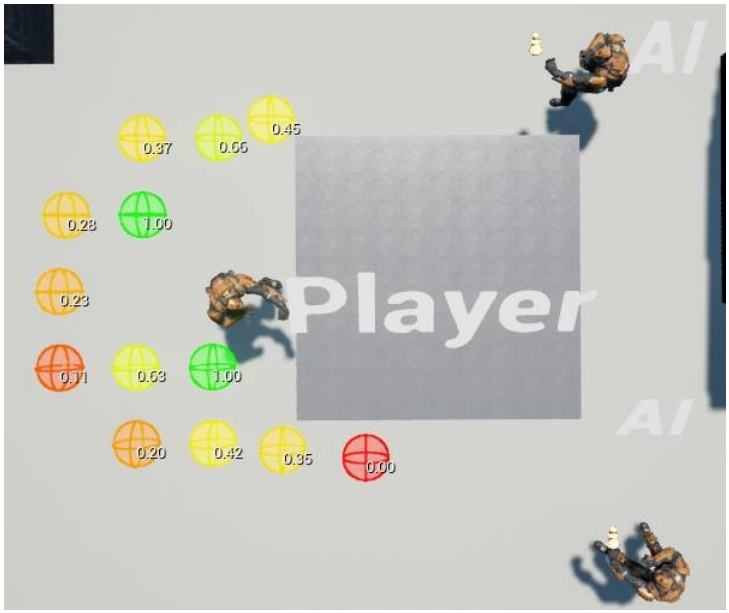


Figura 5.20: Resultado de una consulta EQS para obtener los puntos de supresión

5.4.2.4.7 Punto con linea de fuego

En algunas circunstancias (ver apartado 6.5) nos podemos encontrar con que el agente no tiene una linea clara de disparo para atacar al jugador. Para solucionar este problema también hemos utilizado una consulta *EQS* con dos tests:

1. Test de visibilidad (ver apartado 6.6.1.3). Sólo nos interesa evaluar aquellas posiciones que son visibles para el agente y que puede atacar. Por esta razón, eliminaremos todas las posiciones a las que no tengamos visibilidad.
2. Test de distancia (ver apartado 6.6.1.2). Con este test favorecemos aquellas posiciones que estén más cerca del agente. Es decir, que la distancia sea más baja.

El resultado de esta consulta lo podemos ver en la Figura 5.21 donde las mejores posiciones son más verdes (puntuaciones cercanas a 1). De esta manera, cuando el agente no tenga una visión clara del objetivo, se moverá ligeramente a la izquierda o a la derecha para encontrar una nueva posición desde la cual poder atacar.

Figura 5.21: Resultado de una consulta EQS para obtener nuevos puntos con linea de tiro

06

Implementación

6 Implementación

6.1 Introducción

Para llevar a cabo este proyecto, hemos tenido que mirar el código fuente de *Unreal* [35], definir diferentes clases, sobrescribir métodos, etc. No obstante, creemos que no resulta relevante para la memoria que se comenten estos aspectos y por eso nos centraremos únicamente en la implementación de los métodos más complejos.

6.2 Visibilidad

La visibilidad es la zona que un personaje (jugador o *NPC*) ve. Este algoritmo es totalmente imprescindible y lo utilizamos en diversas ocasiones:

- Para calcular la zona visible por el jugador y determinar las áreas peligrosas y poder realizar el *Tactical Pathfinding* (ver apartado 6.3), es necesario crear un test para determinar las zonas de cobertura (ver apartado 6.6.1.3), etc.
- Para calcular las zonas visibles por los *NPCs* e impedir que la influencia se propague en el mapa de influencia (ver apartado 6.4).

Por lo tanto es necesario que este cálculo sea lo más rápido y eficiente posible ya que se ejecutará de manera frecuente.

A continuación explicaremos dos algoritmos, basados en el trazado de rayos, que se implementaron para calcular la visibilidad [36].

6.2.1 Algoritmo 1

El primer algoritmo, más intuitivo y obvio, es el de trazar un rayo cada un cierto ángulo para determinar las zonas que son visibles y las que no (ver Figura 6.1).

El problema de esta técnica resulta evidente: para obtener una cierta precisión se necesita un número elevado de rayos y, por lo tanto, la eficiencia del algoritmo empeora de manera muy rápida. Aunque no hemos tenido ningún problema de eficiencia relacionado con el numero de rayos en *UE4*, esta solución no parecía ni elegante ni eficiente. Por esta razón, se implementó el algoritmo 2 que permite obtener el mismo resultado pero utilizando un número de rayos mucho menor.

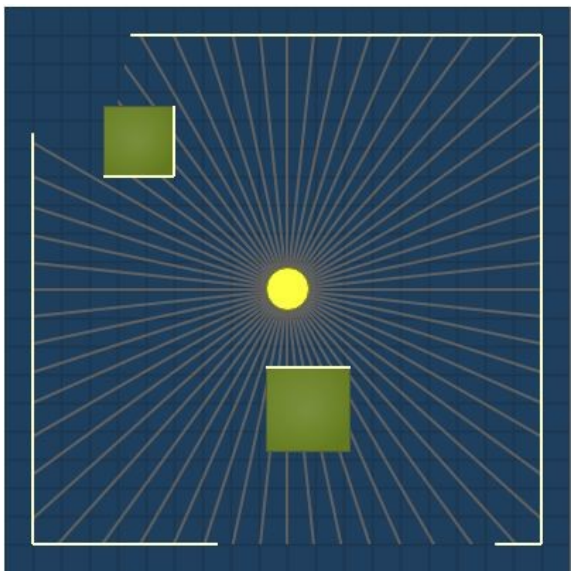


Figura 6.1: Algoritmo para determinar la visibilidad haciendo Raytrace cada un cierto ángulo

6.2.2 Algoritmo 2

La diferencia de este algoritmo respecto del anterior es que, en vez de trazar un rayo cada un cierto ángulo, únicamente se trazan rayos a los vértices de los obstáculos. De esta manera reducimos de manera significativa el número de rayos necesarios. Una vez hemos trazado los rayos, al igual que antes, simplemente determinamos la visibilidad a partir de los triángulos que se forman (ver Figura 6.2).

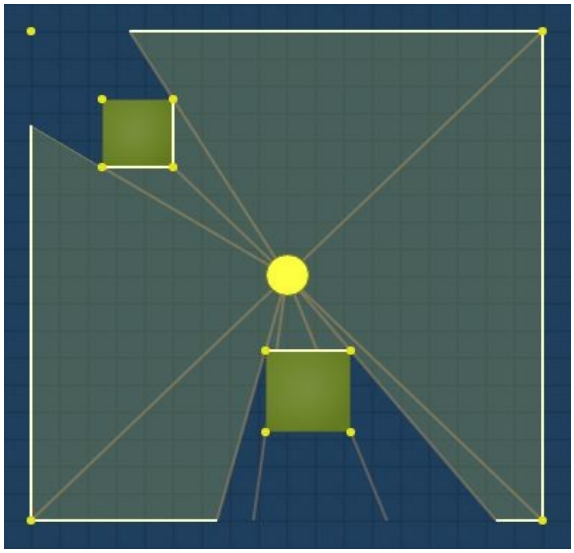


Figura 6.2: Algoritmo para determinar la visibilidad haciendo Raytrace unicamente a los vértices

Con esta idea tan básica en mente, se procedió a implementar el algoritmo con una única diferencia. La visibilidad sólo se calcula en el área visible del agente, es decir, en el área determinada por el rango y ángulo de visión (ver Figura 6.4). En esta zona de visibilidad calcularemos exactamente qué zonas son visibles o no dependiendo de los obstáculos (ver Figura 6.3).

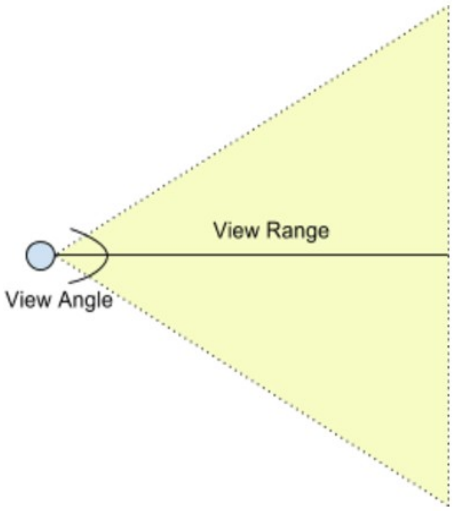


Figura 6.4: Zona de visibilidad por un personaje determinada por el rango y angulo de visión

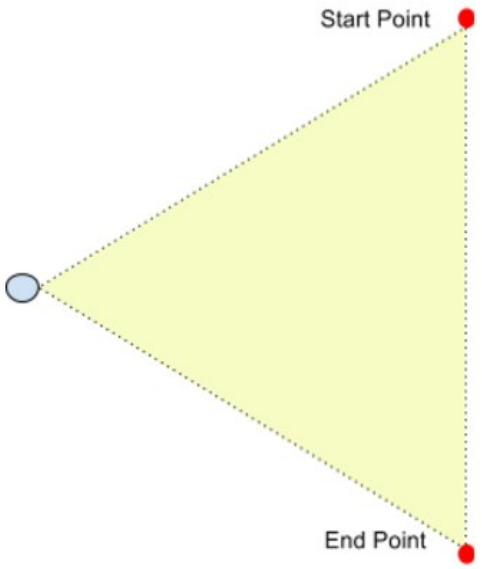


Figura 6.3: Inicio y fin de la zona de visibilidad

Después obtendremos también los vértices de todos los objetos u obstáculos dentro de esta zona de visibilidad. Para asegurarnos que la posterior unión de los triángulos de visibilidad se hace correctamente, es importante que estos vértices estén ordenados tomando como referencia el inicio y el final de la zona de visibilidad (ver Figura 6.5).

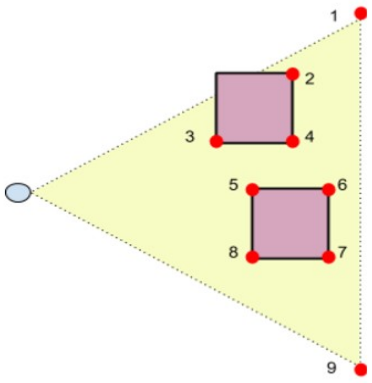


Figura 6.5: Puntos delimitadores de visibilidad ordenados

Ordenados los puntos, trazaremos un rayo a cada vértice para determinar su visibilidad. Al realizar este trazado nos podemos encontrar con diversas situaciones dependiendo del vértice y del resultado al trazar el rayo.

Comenzaremos determinando la visibilidad del primer punto, ya que requiere un tratamiento especial.

Si al trazar el primer rayo, este colisiona con un objeto, nos encontramos con la **situación 1** (ver Figura 6.6). En tal caso, guardaremos el punto de impacto ya que lo usaremos para formar el primer triangulo de visibilidad.

Si por el contrario, el primer rayo no colisiona con ningún objeto, nos encontraremos en la **situación 2** y el punto que guardaremos sera el final de rayo (ver Figura 6.7).

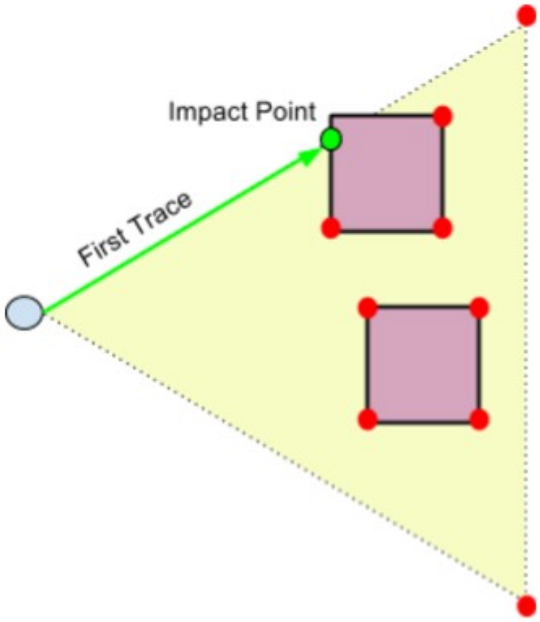


Figura 6.6: Situación 1. Al trazar el primer rayo encontramos una colisión con un obstáculo

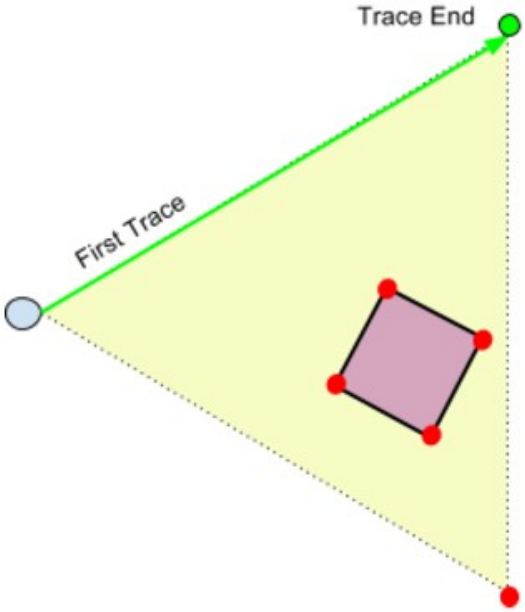


Figura 6.7: Situación 2. Al trazar el primer rayo no hay colisión

Una vez determinada la visibilidad del primer punto, debemos comprobar la visibilidad del resto. A partir de este momento, con cada nuevo rayo podremos construir un triángulo de visibilidad. Este triangulo estará determinado por:

- Vértice 1. Localización del personaje. Es fija y no cambia durante el cálculo.
- Vértice 2. Punto previo determinado por el rayo anterior.
- Vértice 3. Punto actual determinado por el rayo actual.

A medida que vayamos trazando los diferentes rayos el vértice 2 y 3 se modificaran e iremos construyendo los distintos triángulos de visibilidad.

No obstante, durante el trazado de rayos y antes de construir los triángulos, nos podemos encontrar con varias situaciones diferentes que vale la pena destacar.

Si el rayo no es el primero e impacta antes de alcanzar el vértice, nos encontraremos en la **situación 3** (ver Figura 6.8). En este caso, el punto actual (vértice 3) del triangulo vendrá determinado por el punto de impacto del rayo actual y en la siguiente iteración, este punto pasara a ser el vértice 2 del siguiente triángulo.

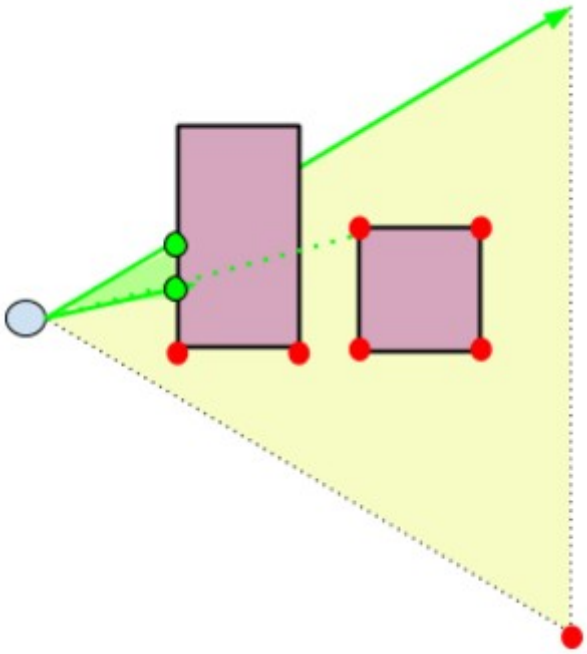


Figura 6.8: Situación 3. El rayo impacta antes de alcanzar el vértice. Triangulo de visibilidad determinado en verde

Si el rayo no es el primero y este no impacta en el vértice directamente, nos encontraremos en la **situación 4** (ver Figura 6.9). Esta situación es especial, ya que en vez de generar un único punto como en las situaciones anteriores, aquí se generan dos:

- El final del rayo se utilizará como tercer vértice del triangulo de visibilidad actual.
- El vértice actual, se utilizara como segundo vértice en el siguiente triangulo.

La **situación 5** es muy parecida que la situación 4 (ver Figura 6.10). La única diferencia, es que los puntos que genera se utilizan de manera inversa, es decir:

- El final del rayo se utilizará como segundo vértice en el siguiente triangulo.
- El vértice actual, se utilizara como tercer vértice del triangulo de visibilidad.

La manera de diferenciar estas dos situaciones es la localización del vértice. Mientras en la situación 4 es el primer vértice del obstáculo, en la situación 5, es el último.

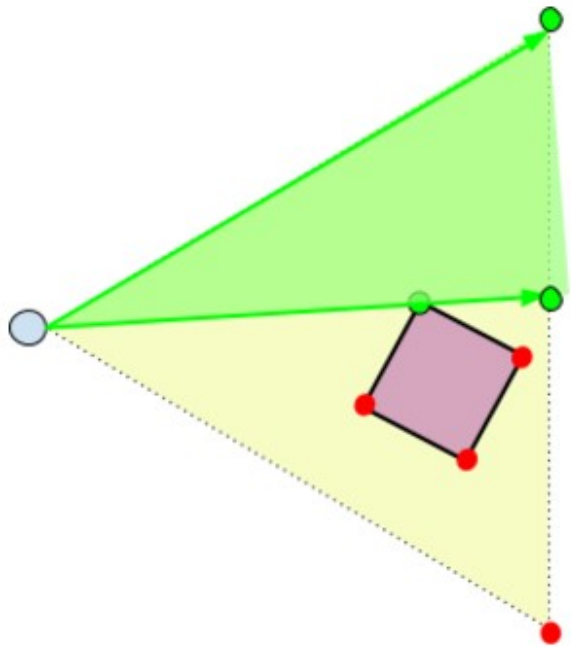


Figura 6.9: Situación 4 Al trazar un rayo no impactamos en el vértice. Triangulo de visibilidad determinado en verde

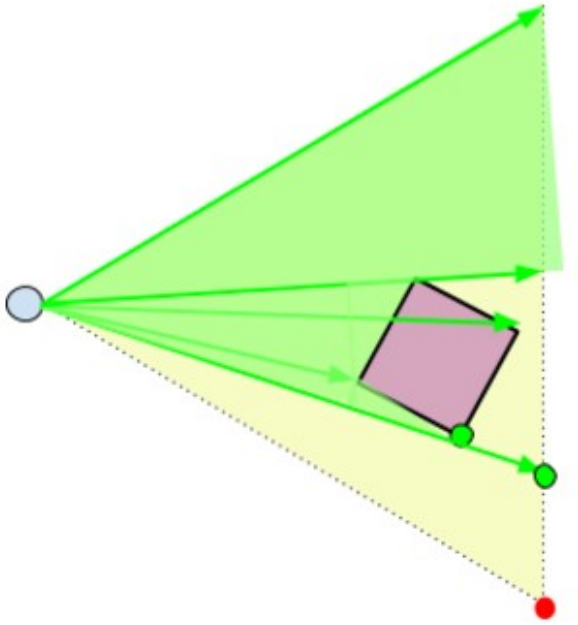


Figura 6.10: Situación 5 Al trazar el un rayo no impactamos en el vértice. Triangulo de visibilidad determinado en verde

Para determinar la visibilidad final del personaje simplemente trazaremos todos los rayos a los vértices teniendo en cuentas las situaciones explicadas anteriormente (ver Figura 6.11) y de esta manera seremos capaces de construir todos los triángulos de visibilidad final (ver Figura 6.12).

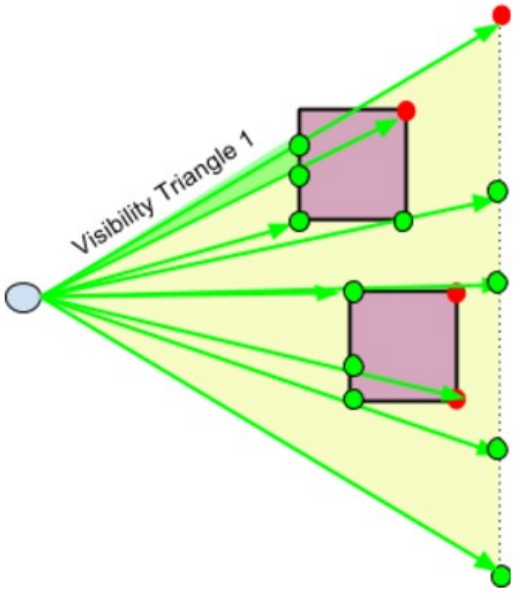


Figura 6.11: Trazado de rayos y construcción de los triángulos de visibilidad

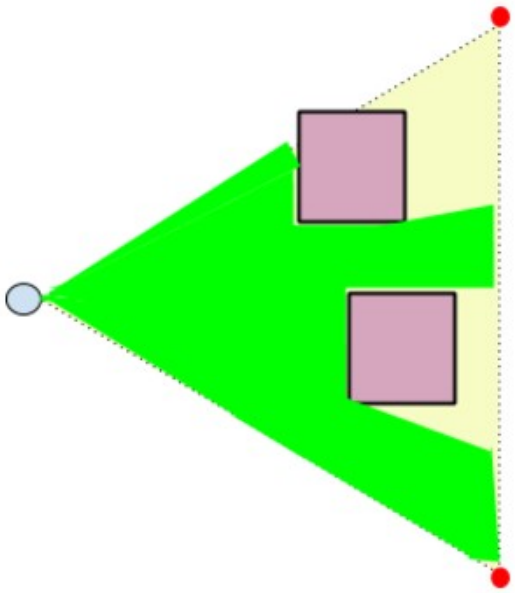


Figura 6.12: Visibilidad final determinada por la unión de los triángulos

Una vez explicado el algoritmo a alto nivel y las situaciones más peculiares, será mucho más fácil entender el pseudocódigo (ver Figura 6.13). Antes que nada, tenemos que preparar las estructuras de datos. Creamos la zona de visibilidad determinada por el rango y angulo de visión (lineas 7-8). Guardamos todos los vértices de los obstáculos de esta área de visibilidad, incluidos el inicio y el fin del área y los ordenamos (lineas 10-16). Después llamamos a una función que será la encargada de determinar los triángulos exactos de visibilidad (lineas 19-20).

```
1  Array<Triangle> CalculateVisibility(Vector Location,
2                                      Orientation Orientation,
3                                      float ViewAngle,
4                                      float ViewRange){
5
6      // Calculate where the visibility starts and where ends
7      Vector EndOfFirstTrace  = Location + Orientation.RotateYaw(-ViewAngle) * ViewRange;
8      Vector EndOfLastTrace   = Location + Orientation.RotateYaw( ViewAngle) * ViewRange;
9      // Get the vertexs of all obstacles in front of us and inside the vision angle and range
10     Array<Vector> VertexsOfObstacles =
11         GetVertexsOfObstacles(Location, Orientation, ViewAngle, ViewRange);
12     // Add the end of the first trace and the end of the last trace as two other vertexs
13     VertexsOfObstacles.Add(EndOfFirstTrace);
14     VertexsOfObstacles.Add(EndOfLastTrace);
15     // Ensure that the vertexs are ordered by the angle starting at the first trace
16     SortByAngle(VertexsOfObstacles, Location, EndOfFirstTrace)
17     // Now we have all the vertexs and their are ordered. Lets join the triangles their form
18     // Calculate Visible Triangles
19     Array<Triangle> VisibleTriangles =
20         CalculateVisibleTriangles(Location, Orientation, ViewAngle, ViewRange, VertexsOfObstacles);
21     return VisibleTriangles;
22 }
```

Figura 6.13: Código del algoritmo para calcular la visibilidad

Esta función itera los diferentes puntos y traza un rayo a cada uno de ellos. Después comprueba en que situación se encuentra (1, 2, 3, 4 ó 5) y crea los triángulos de visibilidad (ver Figura 6.14).

```
Array<Triangle> CalculateVisibleTriangles(Vector Location,
                                          Orientation Orientation,
                                          float ViewAngle,
                                          float ViewRange,
                                          Array<Vector> VertexsOfObstacles){
    Array<Triangle> VisibleTriangles;

    Vector PreviousVertex = null;
    foreach (Vertex CurrentVertex in VertexsOfObstacles){
        // Trace a Ray from the Start location to the CurrentVertex
        HitResult Result = RayTrace(Location, CurrentVertex);
        // Only first trace
        if (PreviousVertex == null){
            // Situation 1 - Hit
            if (Result.BlockingHit){
                PreviousVertex = Result.ImpactPoint;
            }
            // Situation 2 - No hit
            else {
                PreviousVertex = Location + (CurrentVertex - Location)  * ViewRange;
            }
        }
        // All traces except first one
        else {
            // First of all construct the visibility triangle
            Triangle triangle;
            triangle.V2 = Location;
            triangle.V3 = PreviousVertex;

            // Situation 3 - Hit
            if (Result.BlockingHit){
                triangle.V1 = Result.ImpactPoint;
                PreviousVertex = triangle.V1;
            }else {
                // Situation 4 - No hit & leftMostVertex
                if (leftmostVertex(CurrentVertex)){
                    triangle.V1 =  Location + (CurrentVertex - Location)  * ViewRange;
                    PreviousVertex = CurrentVertex;
                }
                // Situation 5 - No hit & rightMostVertex
                else{
                    triangle.V1 = CurrentVertex;
                    PreviousVertex =  Location + (CurrentVertex - Location)  * ViewRange;
                }
            }
            VisibleTriangles.Add(triangle);
        }
    }
    return VisibleTriangles;
}
```

Figura 6.14: Código de la función auxiliar que determina los triángulos de visibilidad

Finalmente, podemos ver que el resultado es muy satisfactorio y permite determinar la visibilidad de manera correcta con un número bajo de rayos (ver Figura 6.15).

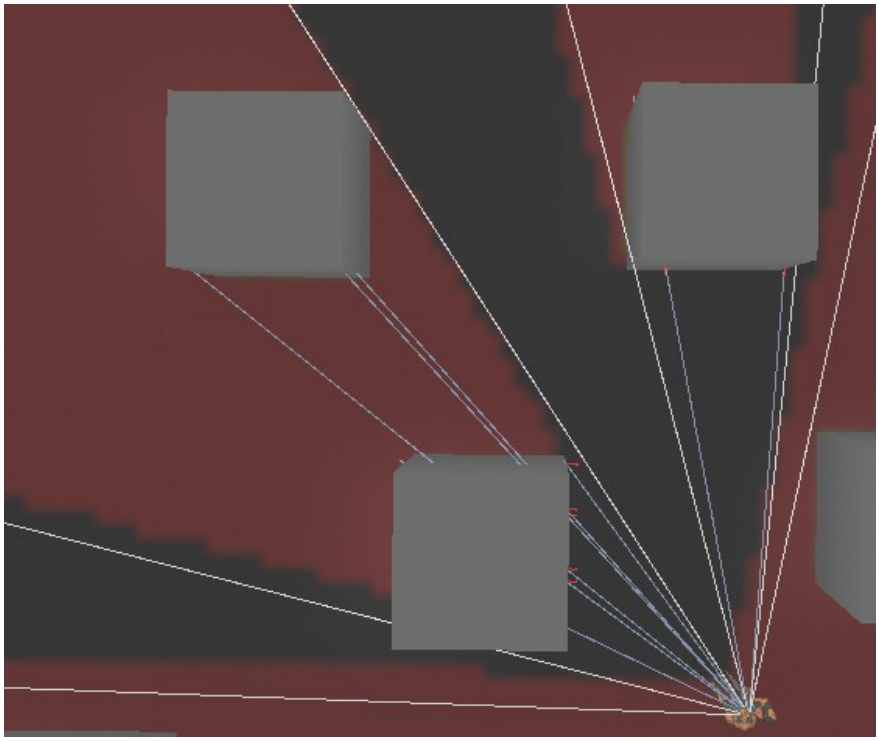


Figura 6.15: Visibilidad determinada (color negro) trazando unicamente rayos a los vértices de los obstáculos

6.3 Tactical pathfinding

Con el fin de cumplir uno de los objetivos propuestos y permitir que los agentes se desplacen mediante caminos seguros decidí implementar el *Tactical pathfinding.*

Para ello, decidimos modificar la heurística original, la cual sólo tenía en cuenta la distancia euclidiana (ver Figura 6.16), para que tuviese en cuenta la seguridad o el peligro de un camino (ver Figura 6.17) y así permitir a los *NPCs* desplazarse de manera segura.

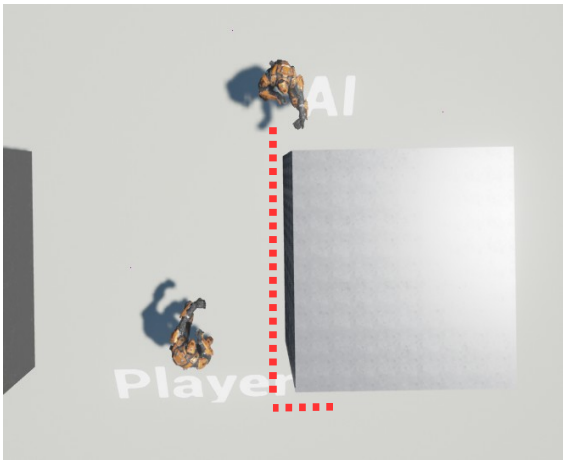


Figura 6.16: Camino más corto determinado por un algoritmo A con una heurística basada unicamente en distancia euclidiana*

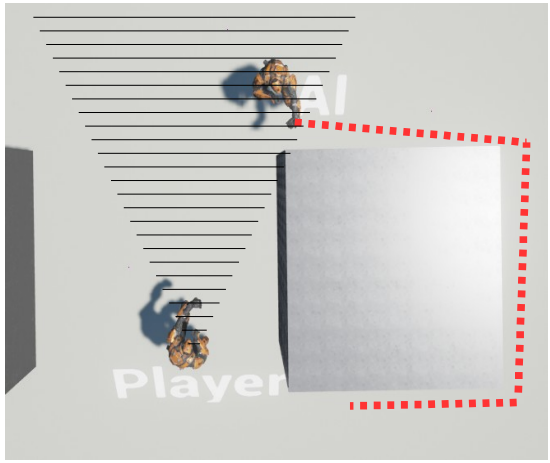


Figura 6.17: Camino más seguro determinado por un algoritmo A con una heurística basada en distancia euclidiana y visibilidad del jugador*

Como podemos observar, un camino es más seguro (menos costoso) si evita zonas visibles por el jugador (calculadas con el algoritmo de visibilidad explicado previamente). Contrariamente, es menos seguro (más costoso) si atraviesa zonas visibles por el jugador.

Por estas razones procedimos a modificar la malla de navegación de manera que cuando el jugador fuese visible por los *NPCs* el coste del algoritmo de *pathfinding* tuviese en cuenta la visibilidad del jugador.

Por lo tanto, es evidente la necesidad de calcular si un camino o simplemente un segmento es o no visible por el jugador. Para ello, implementaremos dos formas de hacerlo, que comentaremos a continuación: algoritmo 1 y algoritmo 2.

6.3.1 Algoritmo 1

La primera opción, y probablemente la mas intuitiva, es buscar la intersección entre la visibilidad y el camino o segmento. La principal ventaja de esta técnica es que es muy precisa ya que nos permite saber exactamente dónde empieza, dónde acaba y cuánto dura la intersección o intersecciones.

No obstante, su principal inconveniente es que su implementación es costosa y su tiempo de ejecución considerable. Recordemos que este coste se utilizara para la función heurística del *A** por lo que no tiene que ser precisa al cien por cien. Ademas, este tipo de evaluaciones se ejecutan de manera muy frecuente y en grandes cantidades por lo que no podemos permitirnos un algoritmo que pueda afectar al rendimiento global del juego.

6.3.2 Algoritmo 2

La idea principal de este algoritmo consiste en dividir el camino o segmento original en un conjunto de puntos, determinar la visibilidad de cada uno de estos puntos (ver Figura 6.18) y por último calcular el coste final a partir de una formula que tiene en cuenta la visibilidad y distancia de todos estos puntos.

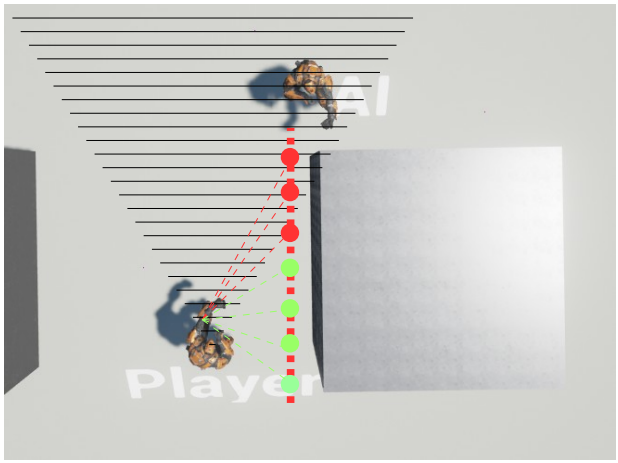


Figura 6.18: Algoritmo 2 para realizar el cálculo del Tactical Pathfinding. Segmento divido en varios puntos. En rojo los puntos más costosos (visibles) y en verde los menos.

Por lo tanto, vemos que este algoritmo como mínimo tiene tres partes: división de segmento o camino en un conjunto de puntos, visibilidad de un punto concreto y cálculo de coste final.

6.3.2.1 División del segmento

Esta división consiste en obtener un conjunto de puntos equidistantes que formen parte del segmento. Originalmente un segmento esta representado por dos puntos, el inicio y el fin. Dado que dependiendo del segmento, dos puntos no son suficientes para determinar su visibilidad, decidimos dividirlo en *N* nuevos puntos equidistantes (ver Figura 6.19) y posteriormente comprobar la visibilidad de todos ellos.

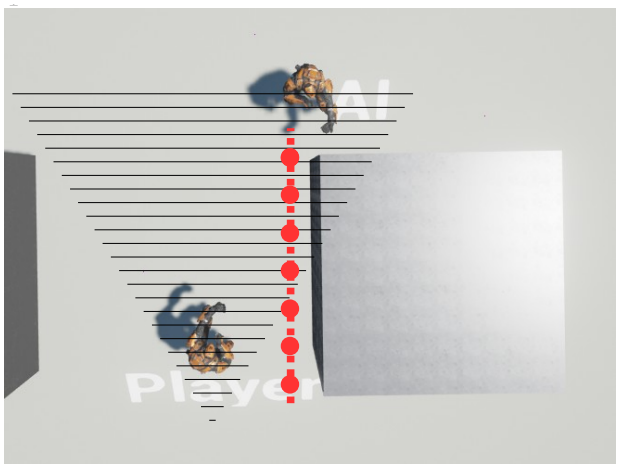


Figura 6.19: División de un segmento en N nuevos puntos equidistantes

El funcionamiento de este algoritmo es muy sencillo. La idea es partir del punto inicial e ir desplazándolo poco a poco sobre el segmento hasta llegar al final. En cada desplazamiento que hagamos, determinaremos un nuevo punto (ver Figura 6.20).

```
1  PuntoAnterior = InicioSegmento;
2  PuntoActual = InicioSegmento;
3  While (PuntoActual != FinalSegmento){
4      DesplazamientoMinimo = (FinalSegmento - InicioSegmento) / NumeroDesplazamientos;
5      PuntoActual = PuntoAnterior + DesplazamientoMinimo;
6      // Calcular visibilidad del punto
7      // Calcular coste
8      PuntoAnterior = PuntoActual;
9  }
```

Figura 6.20: Código del algoritmo de división de un segmento

Como podemos ver, la variable *NumeroDesplazamientos* es la que determina el número de puntos en el cual dividiremos el segmento. Un *NumeroDesplazamientos* muy alto podría afectar negativamente el rendimiento así como un *NumeroDesplazamientos* muy bajo podría resultar poco preciso.

En la implementación, el número de desplazamientos depende del módulo del segmento tal que *NumeroDesplazamientos = ModuloVector * 10.* De esta manera, los segmentos más largos están divididos en más puntos y los cortos en menos. La multiplicación por la constante 10 es simplemente para reducir el número de puntos y agilizar los cálculos.

6.3.2.2 Visibilidad de un punto

Afortunadamente, determinar la visibilidad de cada uno de los puntos obtenidos anteriormente es extremadamente fácil gracias a que representamos la visibilidad de un personaje como un conjunto de triángulos (explicado en el apartado anterior, ver

apartado 6.2). Basta con comprobar si el punto en cuestión esta dentro de alguno de los triángulos de visibilidad utilizando las coordenadas baricéntricas, siempre que los *NPCs* vean al jugador y sean capaces de determinar su visibilidad [37].

6.3.2.3 Cálculo de coste final

Una vez determinada la visibilidad del punto, sólo nos falta determinar su coste. Este viene principalmente determinado por su visibilidad: si el punto es visible por el jugador será más costoso y si no lo es, no lo sera tanto; y por la distancia.

Probablemente, la manera más fácil de realizar este cálculo sea multiplicar el coste del camino por una constante si el punto es visible y por otra si no lo es (ver Figura 6.21)

```
1  PuntoAnterior = InicioSegmento;
2  PuntoActual = InicioSegmento;
3  Coste = 0;
4  While (PuntoActual != FinalSegmento){
5      DesplazamientoMinimo = (FinalSegmento - InicioSegmento) / NumeroDesplazamientos;
6      PuntoActual = PuntoAnterior + DesplazamientoMinimo;
7      // Determinar visibilidad
8      if (EsVisible(PuntoActual, VisibilidadJugador)){
9          // Calcular coste punto visible
10         Modificador = CosteVisible;
11     }else {
12         // Calcular coste punto no visible
13         Modificador = CosteNoVisible;
14     }
15     // Calcular coste
16     Cost += Modificador * Distancia(PuntoActual, PuntoAnterior);
17     PuntoAnterior = PuntoActual;
18 }
```

Figura 6.21: Código del algoritmo 2 completo. Cálculo de coste utilizando un modificador

Aunque este funcionamiento es correcto, también es conceptualmente mejorable. El principal problema es que dados dos segmentos de la misma longitud y con la misma distancia visible por el jugador, este algoritmo devolverá el mismo coste. Esto no es del todo correcto. Si uno de los caminos esta expuesto de manera totalmente contínua es mucho más peligroso que si su zona visible esta expuesta de manera más intermitente, aunque la distancia visible final sea la misma.

Por esta razón, implementamos un algoritmo que incrementase de manera exponencial el coste del segmento a medida que aumenta la distancia visible contínua (ver Figura 6.22).

```
PuntoAnterior = InicioSegmento;
PuntoActual = InicioSegmento;
Coste = 0;
CosteContinuoDentroAreaVisible = 0;
While (PuntoActual != FinalSegmento){
    DesplazamientoMinimo = (FinalSegmento - InicioSegmento) / NumeroDesplazamientos;
    PuntoActual = PuntoAnterior + DesplazamientoMinimo;
    // Determinar visibilidad
    if (EsVisible(PuntoActual, VisibilidadJugador)){
        // Calcular coste punto visible
        CosteContinuoDentroAreaVisible += Distancia(PuntoActual, PuntoAnterior) * CosteVisible;
    }else {
        // Calcular coste punto no visible
        if (CosteContinuoDentroAreaVisible > 0){
            // Venimos de un area visible
            Coste += Potencia(CosteContinuoDentroAreaVisible, Exponente);
            CosteContinuoDentroAreaVisible = 0;
        }
        Cost += Distancia(PuntoActual, PuntoAnterior) * CosteNoVisible;
    }
    PuntoAnterior = PuntoActual;
}
if (CosteContinuoDentroAreaVisible > 0){
    // El ultimo trozo es visible
    Coste += Potencia(CosteContinuoDentroAreaVisible, Exponente);
}
```

Figura 6.22: Ccódigo del algoritmo 2 completo final. Cálculo del coste basado en distancia visible contínua.

En este nuevo algoritmo introducimos una nueva variable *Exponente.* Esta variable nos permite controlar que tan peligroso es un camino dada su contínua visibilidad. También es importante destacar que entre las variables *CosteVisible* y *CosteNoVisible* tiene que haber un equilibrio. A veces, aunque el camino sea un poco más peligroso, vale la pena elegirlo sobre un camino mucho más largo.

6.4 *Mapa de influencia*

Como ya comentamos previamente (ver apartado 2.4.3) los mapas de influencias son muy populares y por ello muy utilizados en videojuegos. Concretamente nosotros los utilizaremos para predecir la posición del jugador cuando lo perdemos de vista e iniciemos la búsqueda.

A continuación, analizaremos los aspectos más relevantes de los mapas de influencia: su representación, uso, configuración, implementación y visualización.

6.4.1 Representación

Antes de proceder a la implementación, hay que decidir como representaremos nuestro mapa de influencia teniendo en cuenta dos aspectos muy importantes:

- **Particionamiento espacial.** Tiene que existir una manera fácil y eficiente de dividir el espacio y guardar información (por ejemplo, influencia) en cada partición.
- **Conectividad.** Representa como están conectadas las diferentes particiones espaciales. Estas conexiones permiten que el algoritmo prediga como se propagara la influencia.

Analizando estos aspectos hay diferentes maneras de representar un mapa de influencia dependiendo de las necesidades y limitaciones del juego [17]. Decidimos utilizar una matriz 2D (ver Figura 6.23). Es la técnica más popular, rápida, fácil de implementar y que mejor se adapta a nuestro mapa (pequeño y con diversos obstáculos).

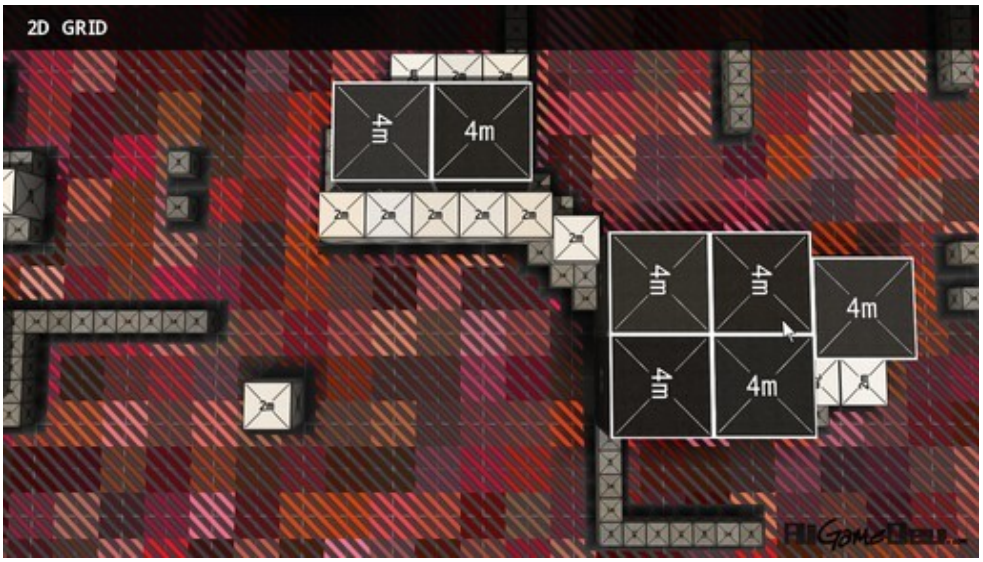


Figura 6.23: Mapa de influencia representado utilizando una matriz 2D

6.4.2 Uso

Como ya hemos comentado en varias ocasiones, el mapa de influencia se utilizará para predecir la posición del jugador. Por lo tanto, cuando los agentes no sepan donde está el jugador, ejecutaran este proceso para predecir su posición. La influencia se comenzará a propagar con el valor máximo en la última posición donde se ha visto al jugador. Las posiciones con valores altos de influencia son posiciones en las que puede estar el jugador con una alta probabilidad. Por último, cuando el jugador es visto nuevamente, la influencia se deja de propagar y el proceso de predicción acaba.

6.4.3 Configuración

Un mismo mapa de influencia puede utilizarse con diferentes objetivos simplemente cambiando tres parámetros que definen su configuración: *momentum, decay* y frecuencia de actualización.

Dado su uso, he procurado ajustar estos parámetros para que la propagación de la influencia se ajuste a la velocidad del personaje. De esta manera, el mapa nos permite predecir de manera muy exacta a que posiciones puede llegar y en que momento.

6.4.3.1 Momentum

Cuando se actualiza un valor de influencia, el *momentum* define cuánto se favorece el nuevo valor respecto del antiguo. Valores de *momentum* altos (alrededor de 1) favorecen los valores antiguos de manera que el mapa guardara datos históricos de otros combates. Por el contrario, valores bajos (cercanos a 0) favorecen la nueva influencia de manera que la propagación sera más rápida y la predicción más acertada.

Para el proyecto, el *momentum* que se ha utilizado es bajo. Esto nos permite predecir la posición del jugador de manera muy acertada.

6.4.3.2 Decay

Este parámetro hace referencia a como debería disminuir o caer la influencia con la distancia. Valores bajos permiten que la influencia se pueda propagar con valores más altos a zonas más lejanas mientras que valores altos provocan que el valor de la influencia disminuya muy rápidamente

En nuestro caso, el valor de *decay* que hemos utilizado es muy bajo (casi 0). Se ha ajustado de tal manera que el valor de la influencia se propaga a la misma velocidad a la que se puede desplazar el jugador.

6.4.3.3 Frecuencia de actualización

El último parámetro que hay que configurar es la frecuencia de actualización que nos

indica cada cuánto tiempo actualizaremos nuestro mapa y sus influencias. Esta frecuencia depende por un lado de nuestros recursos, aunque afortunadamente los mapas de influencia no son nada exigentes; y del uso que daremos al mapa. Mapas a alto nivel se actualizan de poco frecuente, entre 0.5Hz y 1 Hz, mientras que mapas tácticos a bajo nivel se actualizan a 2Hz o 5Hz.

En nuestro caso, la frecuencia de actualización es de 1 Hz.

6.4.4 *Implementación*

```
PropagarInfluencia(){
    // Iteramos todas las posiciones del mapa de influencia
    for (int Index = 0; Index < AlturaMapa * AnchuraMapa; ++Index) {
        float InfluenciaMaxima = 0;
        // Comprovamos si es navegable y no es visible por algun agente
        if (EsNavegable(Index) && !EsVisiblePorAlgunNPC(Index)){
            // Obtenemos sus conexiones
            Array<Index> Conexiones = ObtenerConexionesNavegables(Index);
            // Por cada conexion calculamos la influencia
            foreach (Conexion in Conexiones){
                // Calculamos la influencia a partir de la influencia y la distancia y el decay
                float Influencia = GetInfluencia(Conexion) * exp(-Distancia(Conexion, Index) * DECAY);
                // Nos guardamos el valor maximo
                InfluenciaMaxima = Max(Influencia, InfluenciaMaxima);
            }
            // Calculamos una interpolacion entre el valor antiguo y el nuevo basada en el momentum
            float NuevaInfluencia = Lerp(GetInfluencia(Index), InfluenciaMaxima, MOMENTUM);
            // Modificamos la influencia
            SetInfluencia(Index, NuevaInfluencia);
        }
    }
    ActualizarEstructuraLocal();
}
```

Figura 6.24: Código del algoritmo de propagación de influencia

Por un lado, fue necesario codificar la matriz 2D que representa el mapa de influencia y por el otro, implementar el método de propagación. La matriz 2D , simplemente es una matriz de influencias, por lo que nos centraremos en la implementación del método de propagación (ver Figura 6.24).

Su funcionamiento es muy sencillo y auto-explicativo, aunque vale la pena comentar con más detalle las funciones que se utilizan: *EsNavegable, EsVisiblePorAlgunNPC, ObtenerConexionesNavegables* y *ActualizarEstructuraLocal.*

6.4.4.1 *Función EsNavegable*

Esta función determina si un índice, que al fin y al cabo representa una posición en el mapa, es navegable (forma parte de la *Navmesh).* Aunque hacer esta simple comprobación debería ser fácil accediendo a la *Navmesh* de Unreal, no lo hemos conseguido.

Por esta razón y dado que el mapa es estático decidimos codificarlo como una imagen 2D de baja resolución, donde las partes navegables son de un color y las no navegables de otro (ver Figura 6.25). De esta manera, para comprobar si una

posición es navegable basta con mirar su color en esta imagen.

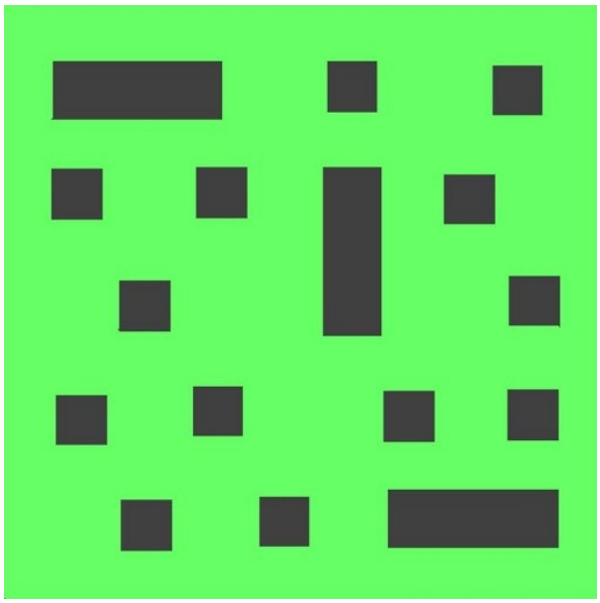


Figura 6.25: Codificación del mapa en una imagen 2D. Las zonas navegables están marcadas en verdes y las no navegables en negro.

6.4.4.2 *Función EsVisiblePorAlgunNPC*

Dado que lo que intentamos predecir con este mapa de influencia es la posición del jugador, tiene sentido que la influencia no se propague en aquellas posiciones que los *NPCs* pueden ver. La influencia solo se propaga en aquellas zonas que los *NPCs* no ven y por tanto, tienen que hacer predicciones.

Para determinar las zonas visibles de cada agente, utilizamos el explicado algoritmo de visibilidad (ver apartado 6.2). Esta función simplemente tiene que determinar si el indice (la posición) es visible o esta dentro de algunos de los triángulos de visibilidad de algún *NPC.*

6.4.4.3 *Función ObtenerConexionesNavegables*

Al utilizar una matriz 2D como estructura para representar el mapa, obtener los vecinos navegables es muy simple. Basta con comprobar las posiciones navegables que estén en contacto directo (alrededor) de la posición dada (ver Figura 6.26).

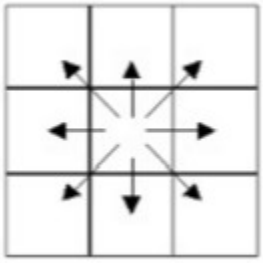


Figura 6.26: Vecinos de una posición en una matriz

6.4.4.4 *Función ActualizarEstructuraLocal*

Por último, para que las influencias se calculen correctamente, hace falta utilizar dos estructuras de datos. La primera sólo se utiliza para leer los valores de influencia del fotograma actual y la otra sólo se utiliza para realizar las modificaciones. En caso contrario, a la misma vez que se modifica la estructura, también se está leyendo de ella, lo que provoca valores y propagaciones incorrectas.

El objetivo de esta función es igualar la estructura de lectura para que esté actualizada una vez ha acabado de propagarse la influencia en la estructura de escritura. De esta manera, se prepara para la siguiente ejecución.

6.5 *Disparos y supresión*

A diferencia de los algoritmos anteriores, los cuales eran, más o menos complejos, el funcionamiento de esta función es bastante simple. No obstante, consideramos relevante analizarla dado el tipo de juego en el que se basa el proyecto (ver Figura 6.27).

```
if (NeedReloadNow){
    Recargar();
}else {
    if (PlayerIsVisible){
        // Disparar
        if (LineaDeFuego(PlayerLocation)){
            Disparar(PlayerLocation);
        }else {
            // Calculamos un nuevo punto una linea de fuego
            PuntoConLineaDeFuego = EQS_PuntoConLineaDeFuego();
            MoveTo(PuntoConLineaDeFuego);
        }
    }else {
        // Supresion
        // Calculamos un punto de supresion
        PuntoDeSupresion = EQS_Supresion();
        // Notificamos que hemos elegido esta posicion de supresion
        SuppressionMap.Add(NombreDelAgente, PuntoDeSupresion)
        Disparar(PuntoDeSupresion);
    }
}
```

Figura 6.27: Código de la función de disparar y suprimir

En este código hay tres funciones que vale la pena analizar más a fondo: *LineaDeFuego*, *EQS_PuntoConLineaDeFuego* y *EQS_Supresión*.

6.5.1 Linea de Fuego

En el momento en el que en el juego agregamos múltiples agentes (entre tres y cuatro) vimos un problema muy claro relacionado con el fuego amigo: cuando los agentes intentaban disparar al jugador se mataban entre ellos (ver Figura 6.28).

Figura 6.28: Situación de combate con un agente AI 2 (señalado en rojo) sufriendo fuego amigo

La manera más simple y elegante de solucionar este problema es que antes, de disparar, los *NPCs* comprueben si hay algún otro agente en su linea de fuego. Su implementación es muy simple (ver Figura 6.29).

```
bool LineaDeFuego(PosicionObjetivo){
    // Trazamos un rayo
    Resultado = RayTrace(MiPosicion, PosicionObjetivo);
    // Si el rayo colisiona y la colision es con un NPC
    if (Resultado.Colision && Resultado.Colision.Clase = NPC){
        // No hay linea de fuego
        return false;
    }
    // En cualquier otro caso
    else {
        // Hay linea de fuego
        return true;
    }
}
```

Figura 6.29: Código de la función de cálculo linea de fuego

6.5.2 EQS: Punto con linea de fuego

En el caso de no tener una linea de fuego clara, utilizaremos la consulta *EQS* definida en el apartado 5.4.2.4.7 para para obtener una nueva posición con una linea de fuego. Gracias a esta función podemos re-posicionar a los agentes, un poco más a la derecha o a la izquierda, para tener una mejor linea de tiro antes de disparar.

6.5.3 EQS: Supresión

Cuando el jugador no es visible, los *NPCs* son capaces de realizar fuego de supresión en las posiciones que el jugador podría utilizar para atacar. Al igual que antes, para determinar estas posiciones lo haremos mediante una consulta *EQS* definida en el apartado 5.4.2.4.6.

6.6 Tests EQS

Para poder implementar las diferentes consultas tácticas *EQS* explicadas en el apartado 5.4.2.4 elaboramos un conjunto de tests que permitiese filtrar o puntuar las posiciones según deseáramos.

Algunos de los tests que explicaremos a continuación son tests genéricos que se pueden y se usan en varias consultas, mientras que otros únicamente se utilizan en una consulta y tienen un objetivo muy específico.

6.6.1 Tests genéricos

6.6.1.1 Test de coste (pathfinding)

Este test permite asignar una puntuación a cada posición de acuerdo al coste calculado por el algoritmo de *pathfinding* desde la posición en la que se encuentra el

personaje que hace la consulta a la posición que se evalúa (ver Figura 6.30). Este test permite tanto favorecer las posiciones cercanas como las lejanas, dependiendo de las necesidades que tengamos.

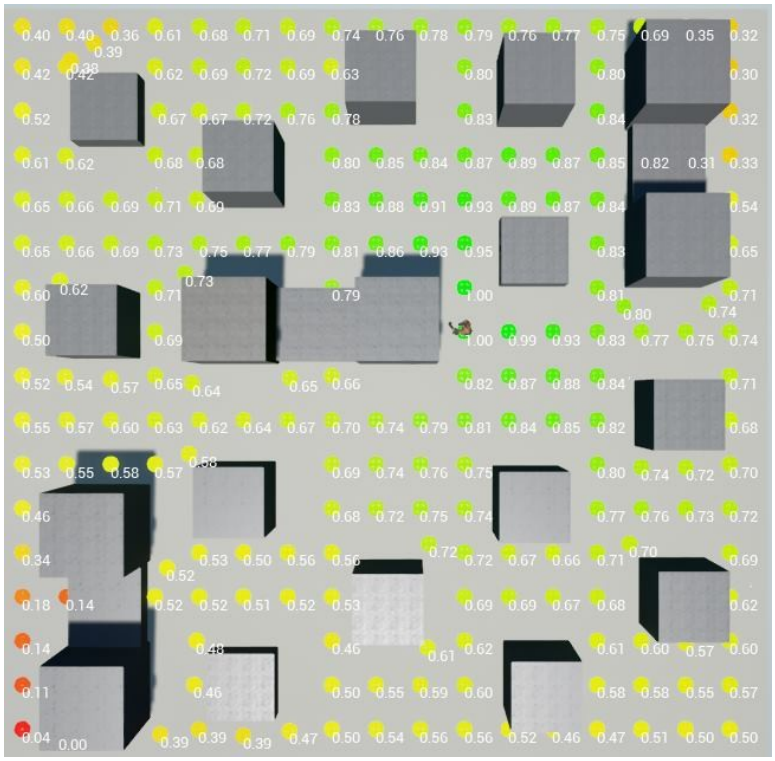


Figura 6.30: Resultado de una consulta EQS con un único test de coste favoreciendo las posiciones cercanas

6.6.1.2 *Test de distancia*

A diferencia del test anterior que utilizaba el coste del algoritmo de *pathfinding* para puntuar las posiciones, éste utiliza la distancia euclidiana.

En la Figura 6.31 vemos el resultado de ejecutar este test desde un personaje favoreciendo aquellas posiciones que se encuentran a una distancia más pequeña. No obstante, al igual que antes, también podríamos ejecutar la consulta siguiendo el razonamiento inverso y puntuar de manera más elevada aquellas posiciones más alejadas.

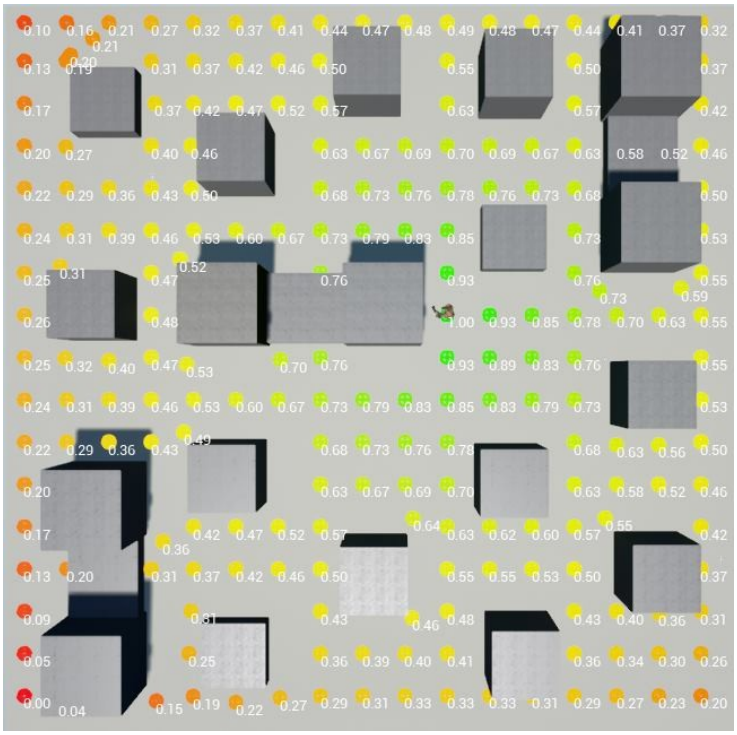


Figura 6.31: Resultado de una consulta EQS con un único test de distancia de un personaje a todos los puntos del mapa

6.6.1.3 *Test de visibilidad*

El objetivo de este test es filtrar aquellas posiciones que son visibles por un personaje (jugador o *NPC*). Como resulta lógico, *Unreal* ya incorpora de manera nativa un test para realizar el cálculo de visibilidad. Tras analizar la implementación vimos que este test traza un rayo a cada posición para determinar su visibilidad. Para evitar realizar un número tan elevado de rayos, decidimos crear un test que re-utilizara nuestro algoritmo previo se visibilidad (ver apartado 6.2).

Este test es muy sencillo. Una vez calculada la visibilidad del jugador con el algoritmo de visibilidad, simplemente comprobamos si las posiciones están dentro de alguno de los triángulos de visibilidad (ver Figura 6.32).

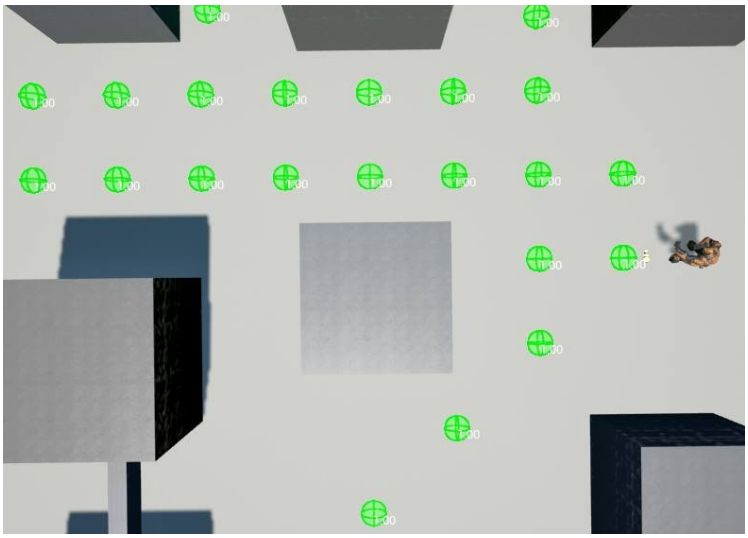


Figura 6.32: Resultado de una consulta EQS con un único test de visibilidad

6.6.1.4 *Test "detrás" de los obstáculos*

Este test tiene dos objetivos. Por un lado elimina todas aquellas posiciones visibles por el jugador y por el otro puntúa las posiciones favoreciendo aquellas que están cerca de un obstáculo en la dirección en la que se encuentra el jugador.

Para ilustrar este razonamiento veamos la Figura 6.33. El punto *p3* queda directamente descartado ya que es visible por el jugador. Los puntos *p1* y *p2* no son visibles por lo que son buenos candidatos. El punto *p1* tiene una puntuación mas grande ya que la distancia *d1* al obstáculo en dirección al jugador es menor que la distancia *d2* desde el punto p2 al obstáculo en dirección al jugador.

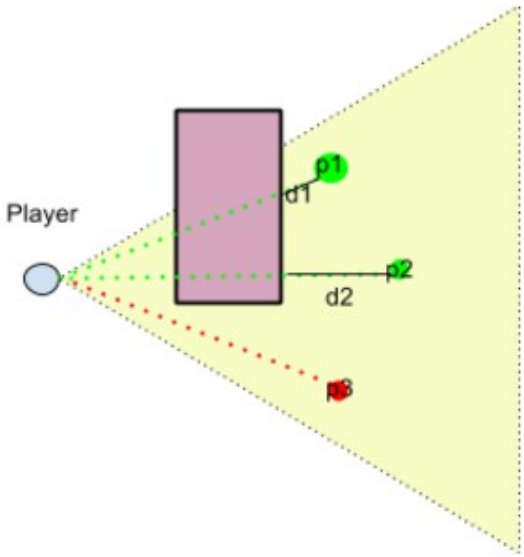


Figura 6.33: Demostración de la formula de puntuación utilizada en el test

Gracias a este razonamiento, no sólo conseguimos posiciones que no son visibles por el jugador si no que también nos aseguramos que la posición está cerca de un obstáculo y, que por lo tanto, es mucho más segura.

El resultado de la consulta lo podemos ver en la Figura 6.34, donde las posiciones no visibles por el jugador y más cercanas a un obstáculo van de color verde (mejor) a rojo (peor).

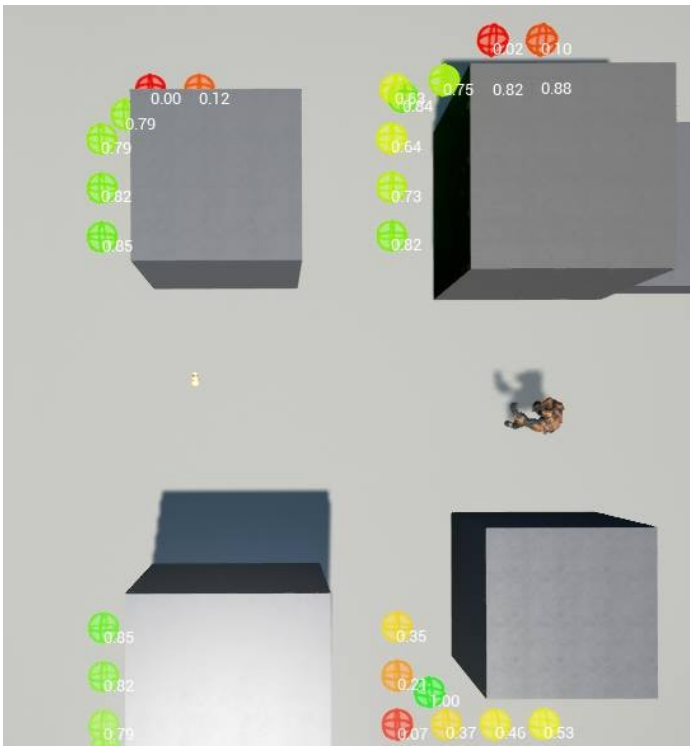


Figura 6.34: Resultado de una consulta EQS con un único test de detrás de los obstáculos

6.6.1.5 Test de proximidad a los obstáculos

Este test es muy parecido al anterior con la diferencia de que no tiene en cuenta la posición del jugador. Este test simplemente puntúa de manera alta aquellas posiciones que están cerca de un obstáculo.

En la Figura 6.35 vemos como aquellas posiciones más cercanas a los obstáculos tienen mejor puntuación que las que están más lejos. Ademas, las posiciones cercanas a más de un obstáculo tienen una puntuación aun mayor que las que lo están sólo de uno.

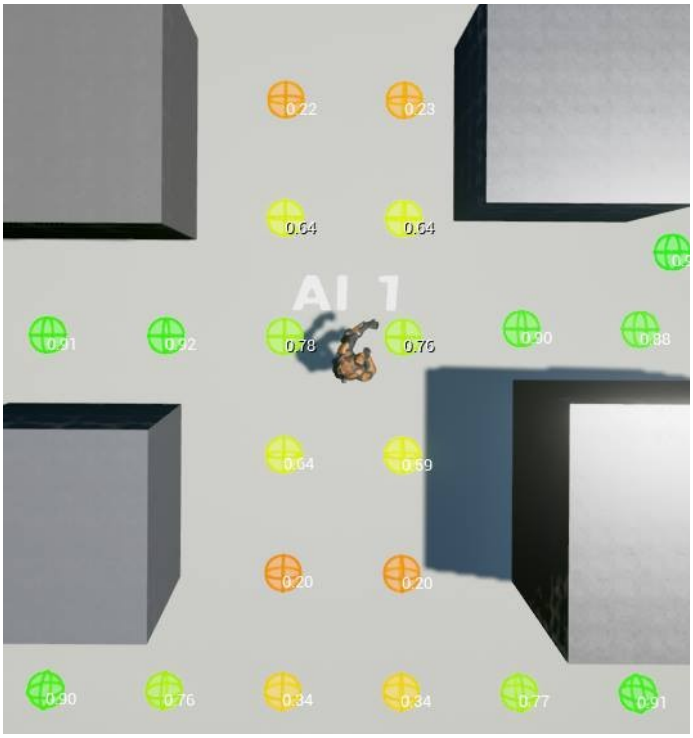


Figura 6.35: Resultado de una consulta EQS con un único test de proximidad a los obstáculos

Gracias a este test somos capaces evitar espacios abiertos sin ninguna cobertura y permitimos a los agentes desplazarse a través de posiciones potencialmente seguras en las que podría cubrirse si hiciese falta.

6.6.1.6 Test de aleatoriedad

Si ejecutamos una única consulta *EQS* desde una misma posición, ésta siempre nos devolverá los mismos resultados. Esto genera situaciones en las que los *NPCs* escogían sistemáticamente la misma posición para atacar y defender una y otra vez. Este comportamiento resulta muy predecible y poco humano.

Por este motivo creamos este test que sirve agregar aleatoriedad a los resultados. Obviamente este test no va a hacer que un buen resultado sea malo o a la inversa. Simplemente, substituye entre un 5% y un 10% de la puntuación de cada resultado por un valor aleatorio delimitado inferior y superiormente. De esta manera, las puntuaciones de los resultados cambiaran ligeramente cada vez que ejecutemos la consulta y por tanto los agentes tomaran diferentes decisiones cada vez, haciéndolos más impredecibles y más “humanos”.

6.6.2 Tests específicos

6.6.2.1 Test de influencia

Este test se utiliza específicamente para el comportamiento de búsqueda y permite filtrar o puntuar las posiciones generadas por la consulta de acuerdo a si están cerca de posiciones con alta influencia en el mapa de predicción (variable *PredictionMap).* Las posiciones con alta influencia son posiciones probables en las que se puede encontrar el jugador (ver Figura 6.36). Más detalles sobre el funcionamiento de los mapas de influencia en el apartado 6.4.

Para el cálculo de la puntuación decidimos utilizar la media de la influencia de los vecinos.

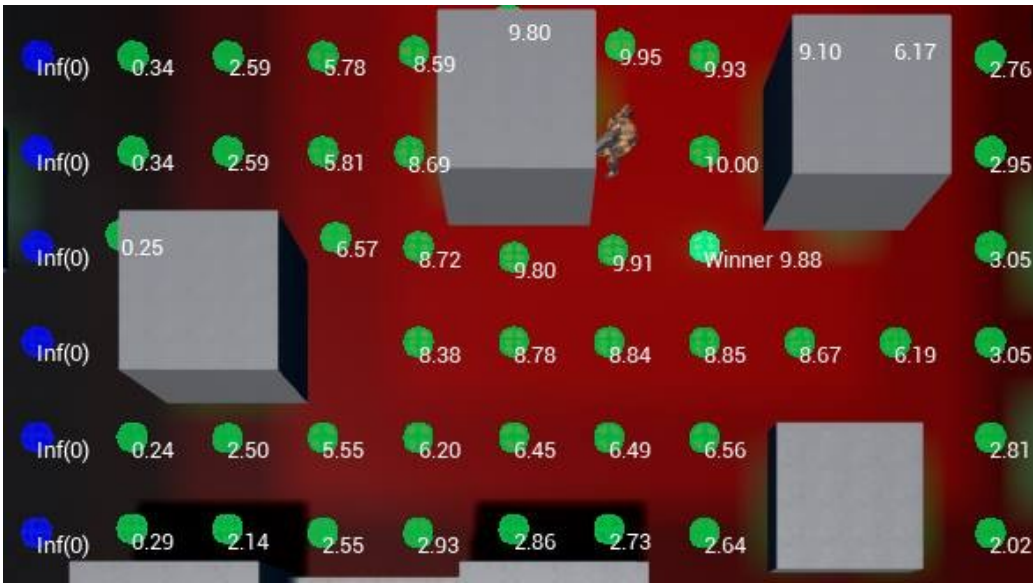


Figura 6.36: Resultado de una consulta EQS con un único test de influencia

6.6.2.2 Test de expansión

Este test también se utiliza únicamente en el comportamiento de búsqueda. Su objetivo es favorecer las posiciones que están más alejadas de un conjunto de posiciones de búsqueda escogidas guardadas en la variable *SearchMap.*

La idea es favorecer aquellas posiciones tal que la distancia mínima que forman con las posiciones de *SearchMap* es grande. En la Figura 6.37 vemos el resultado de ejecutar una consulta con este test. En esta Figura el agente *AI 2* esta evaluando la mejor posición de búsqueda teniendo en cuenta la posición actual del agente *AI 1*. Vemos como las posiciones más cercanas al agente *AI 1* tienen una puntuación baja y a medida que nos alejamos la puntuación aumenta.

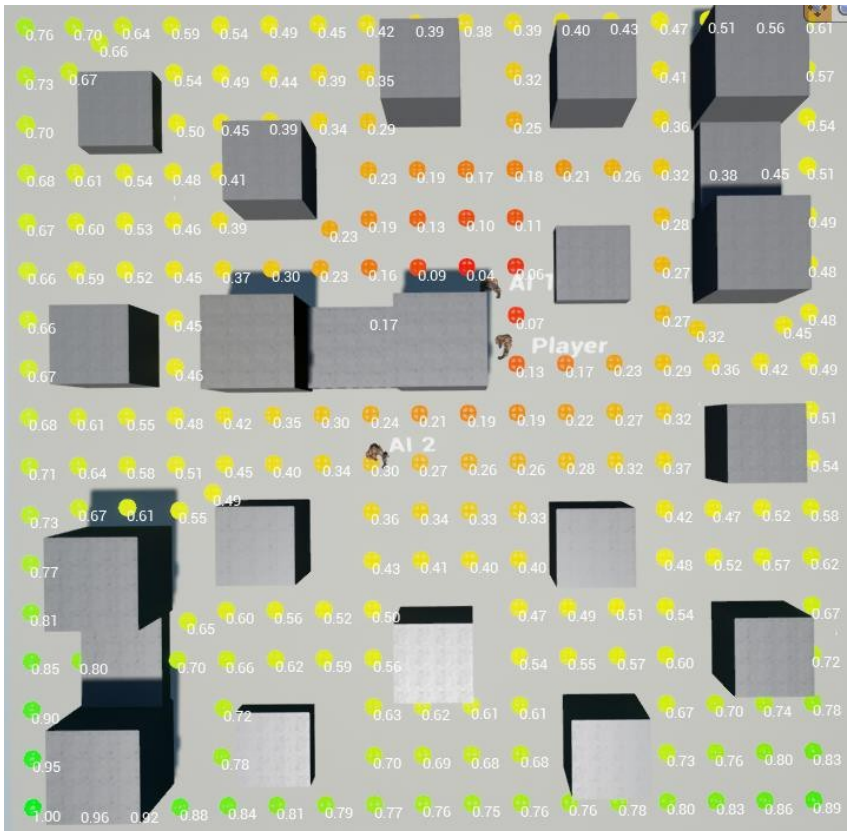


Figura 6.37: Resultado de una consulta EQS con un único test de expansión favoreciendo las posiciones alegadas del agente AI 1

Mediante el uso de este test favorecemos que las posiciones de búsqueda estén alejadas la una de la otra generando una sensación de expansión y distribución y cubriendo el mayor terreno posible, dificultando al jugador la tarea de esconderse.

6.6.2.3 Test de flanqueo

Este test se utiliza para evaluar las posiciones de ataque y avance. De una manera similar al test anterior, el objetivo de este test es favorecer aquellas posiciones que permitan rodear al jugador para atacarlo por diferentes flancos teniendo en cuenta las posiciones de ataque de todos los agentes guardadas en la variable *AttackMap.*

La idea es favorecer aquellas posiciones donde el ángulo mínimo que forman con las posiciones de *AttackMap* es grande. De esta manera puntuamos de manera alta las posiciones de ataque que se encuentren en ángulos lo más diferentes posibles, generando un ataque por diferentes flancos.

En la Figura 6.38 podemos ver que la Posición *1* es mejor que la *Posición 2* ya que el ángulo *A* que forma con la ya escogida P*osición de Ataque 1* es mayor que el angulo *B* que forma la *Posición 2.*

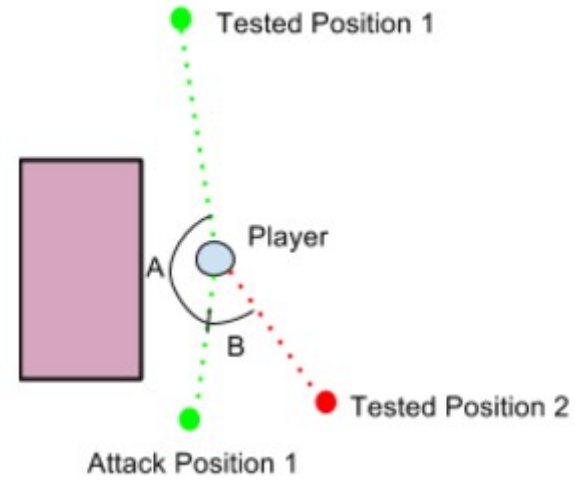


Figura 6.38: Demostración de la formula utilizada en el test de flanqueo

El resultado de implementar este concepto en una consulta EQS lo podemos ver en la Figura 6.39, donde el agente *AI 1* ya ha escogido su posición de ataque al jugador y el otro agente *AI 2* esta evaluando qué posición es mejor para atacar, teniendo en cuenta al agente *AI 1*. Podemos ver que en rojo están puntuadas aquellas posiciones con el angulo más bajo con la posición ya escogida por el agente 1; en amarillo un angulo intermedio y en verde un angulo grande.

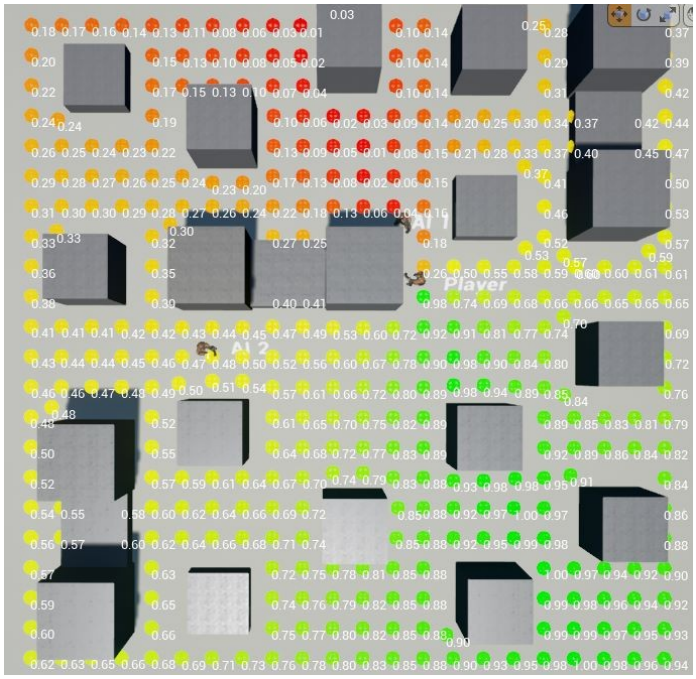


Figura 6.39: Resultado de una consulta EQS con único test de flanqueo

Gracias a este test podemos obtener posiciones de ataque distribuidas en diferentes flancos generando una sensación de comportamiento y coordinación en grupo durante el ataque.

6.6.2.4 Test de repetición

Este test unicamente se utiliza para la consulta relacionada con las posiciones de supresión. Es extremadamente simple y su responsabilidad es eliminar de la consulta aquellas posiciones que estén cerca de alguna posición existente de supresión de algún otro agente. Todas estas posiciones se guardan en la variable *SuppressionMap.*

Gracias a este test evitamos que múltiples agentes supriman una misma posición y los forzamos a buscar otra posición que suprimir. En caso de que no encuentren otra posición que valga la pena suprimir, simplemente no hacen nada.

07

Contratiempos

7 Contratiempos

7.1 Introducción

Durante el desarrollo del proyecto, y sobretodo de la implementación, nos hemos encontrado con diversos problemas relacionados con *Unreal* y también con conceptos de IA. La fuente de la mayoría de estos problemas fue una falta de conocimiento y la ausencia de documentación por parte de *Unreal.*

A continuación explicaremos rápidamente los problemas más interesantes y/o complicados con los que nos encontramos durante el desarrollo del proyecto.

7.2 Mala percepción

El único problema que encontramos con el sistema de percepción es que sólo era capaz de ver al jugador si veía el centro del mismo. Esto provocaba que en algunas situaciones en las que al jugador se le veían los hombros o la cabeza, el agente no lo detectara (ver Figura 7.1). Nos pareció un problema totalmente inadmisible por lo que procedimos a solucionarlo lo antes posible.

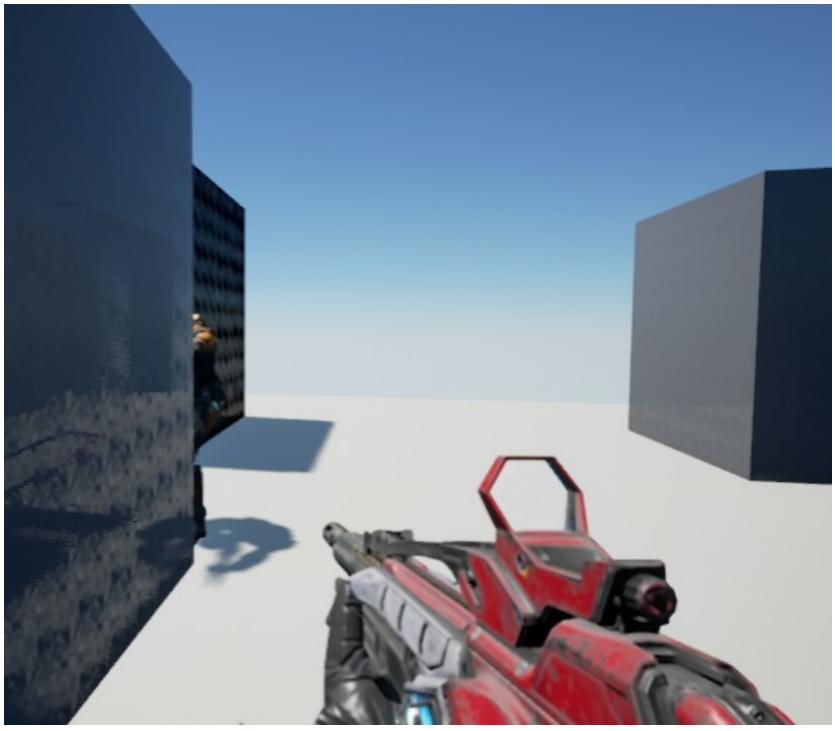


Figura 7.1: Situación problemática en la que la IA no detecta que el jugador es visible

La implementación por defecto de la función de percepción activa el sentido de la vista si existe un rayo que interseque el centro del objeto y los “ojos” del sistema de percepción.

Realizamos unas pequeñas modificaciones a esta función para que, en vez de un rayo al centro, trazase tres rayos a las posiciones más a la derecha, más a la izquierda y más arriba. Estas posiciones se corresponden con el hombro derecho, el hombro izquierdo y la cabeza (ver Figura 7.2) [38].

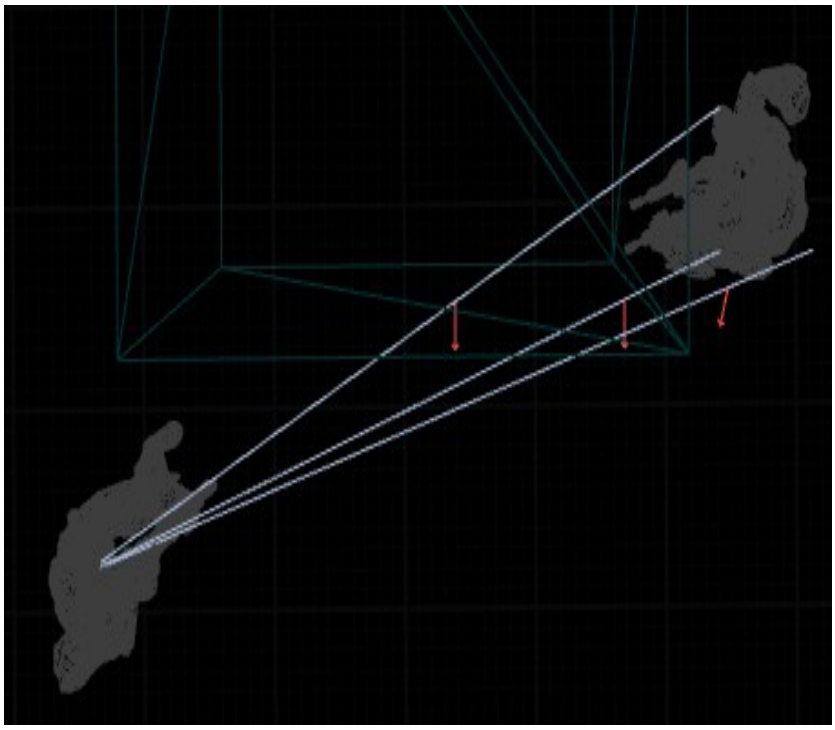


Figura 7.2: Solución utilizada para detectar la visibilidad a través del uso de tres rayos (cabeza y hombros)

De esta manera, el agente es capaz de detectar al jugador si solamente le ve la cabeza o uno de los hombros.

7.3 Colisiones entre agentes

Justo después de solucionar el problema del fuego amigo descubrimos otro problema. Cuando los agentes se tenían que desplazar, colisionaban unos con otros y se quedaban atascados. Sus desplazamientos se volvieron muy torpes, se movían con muchas dificultades y prácticamente nunca llegaban a su objetivo (ver Figura 7.3).

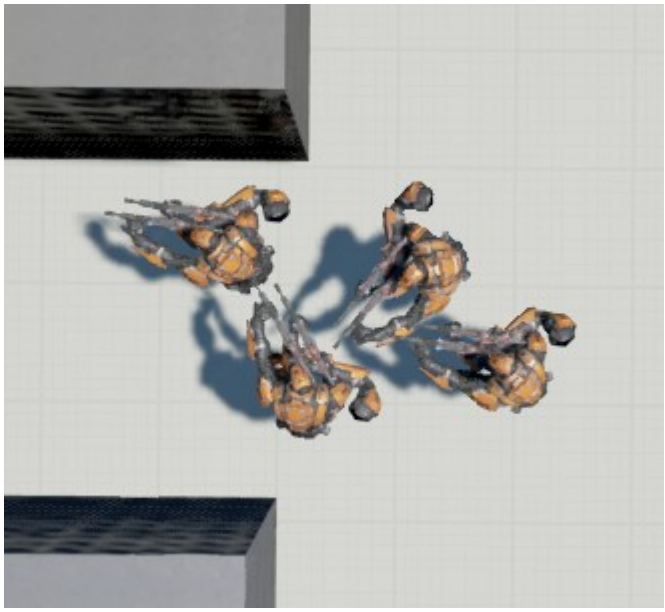


Figura 7.3: Situación problemática en la cual los NPCs se han quedado atascados

Tras una breve investigación averiguamos que este es un problema muy común de la IA. Afortunadamente ya se ha estudiado mucho este tema y los algoritmos que resuelven este problema se llaman *Steering Behaviors [39]*. Su función es implementar de manera sencilla un algoritmo que permita esquivar la colisión de una manera similar a como lo haría un humano.

Por suerte, *UE4* incorpora uno de estos algoritmos que simplemente tuvimos que activar para utilizarlo con nuestros *NPCs [40].* Internamente esta solución se basa en un algoritmo *RVO (Reciprocal Velocity Obstacle) [41].* Básicamente, la idea de este algoritmo es evitar las colisiones de múltiples agentes independientes sin establecer explícitamente una comunicación entre ellos, evitando comportamientos oscilantes. Esto es posible gracias a que cada agente sólo toma la mitad de la responsabilidad para evitar la colisión y asume que el otro agente hará el resto (de aquí la reciprocidad).

El algoritmo *RVO* solo es una mejora del algoritmo *VO (Velocity Obstacle).* En este algoritmo cada agente realiza cálculos para comprobar si en el futuro colisionará con algún otro agente en movimiento y por lo tanto si necesita modificar su velocidad para evitar dichas colisiones.

Su funcionamiento es bastante sencillo (ver Figura 7.4). Dado el agente A en la posición P_A y el obstáculo (en movimiento) B en la posición P_B, la $VO^{AB}(V_B)$ del obstáculo B al agente A es el conjunto de velocidades V_A que resultaran en una colisión en algún momento con el obstáculo B a velocidad V_B.

Si al trazar un rayo desde P_A en la dirección de la velocidad relativa (V_A – V_B) este

interseca con el radio ampliado de B y centrado en el punto P_B podemos decir que la velocidad V_A esta dentro de la velocidad obstáculo de B. De esta manera $VO^{AB}(V_B)$ es simplemente el cono con origen en P_A desplazado V_B, con dirección P_B y su ancho esta determinado por el radio ampliado de B.

Por lo tanto, si V_A esta dentro de $VO^{AB}(V_B)$ podemos asegurar que en algún momento A y B colisionarán. Si por el contrario V_A esta fuera de la velocidad obstáculo de B, los objetos nunca colisionarán.

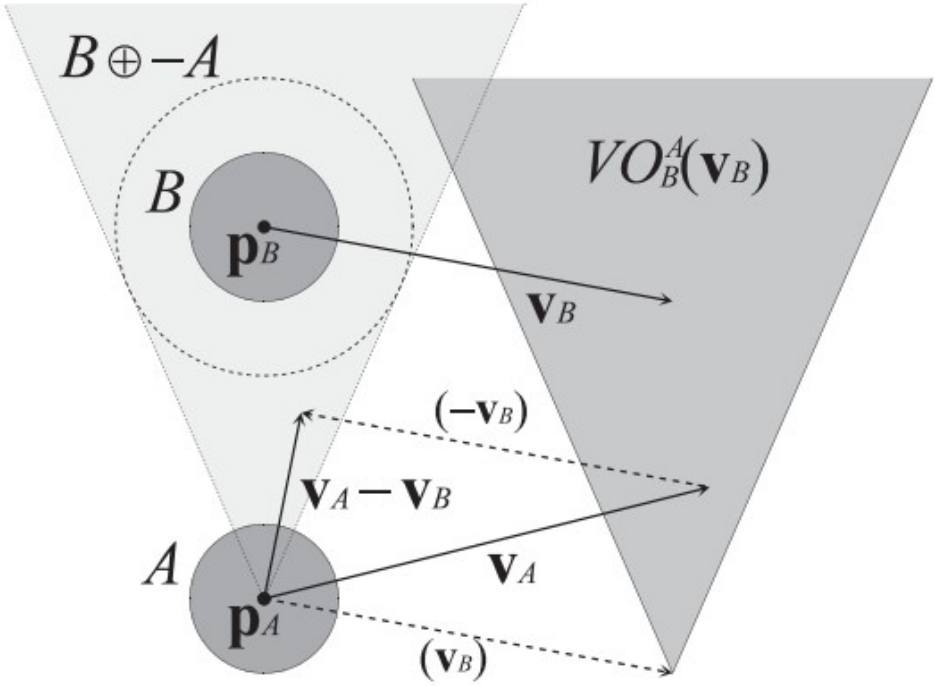


Figura 7.4: VO de un obstáculo B respecto un agente A

Este concepto ha sido ampliamente utilizando para la navegación con múltiples agentes. No obstante, en ciertas ocasiones cuando los agentes modifican su velocidad a la misma vez esta técnica puede presentar movimientos oscilantes ocasionados.

RVO soluciona estos comportamientos ya que $RVO^{AB}(V_B)$ contiene todas las velocidades que son la media entre la velocidad V_A y la velocidad obstáculo $VO^{AB}(V_B)$ (ver 7.5).

Aunque para ilustrar los ejemplos se han utilizado dos agentes, estas técnicas son fácilmente extrapolables a situaciones con múltiples agentes.

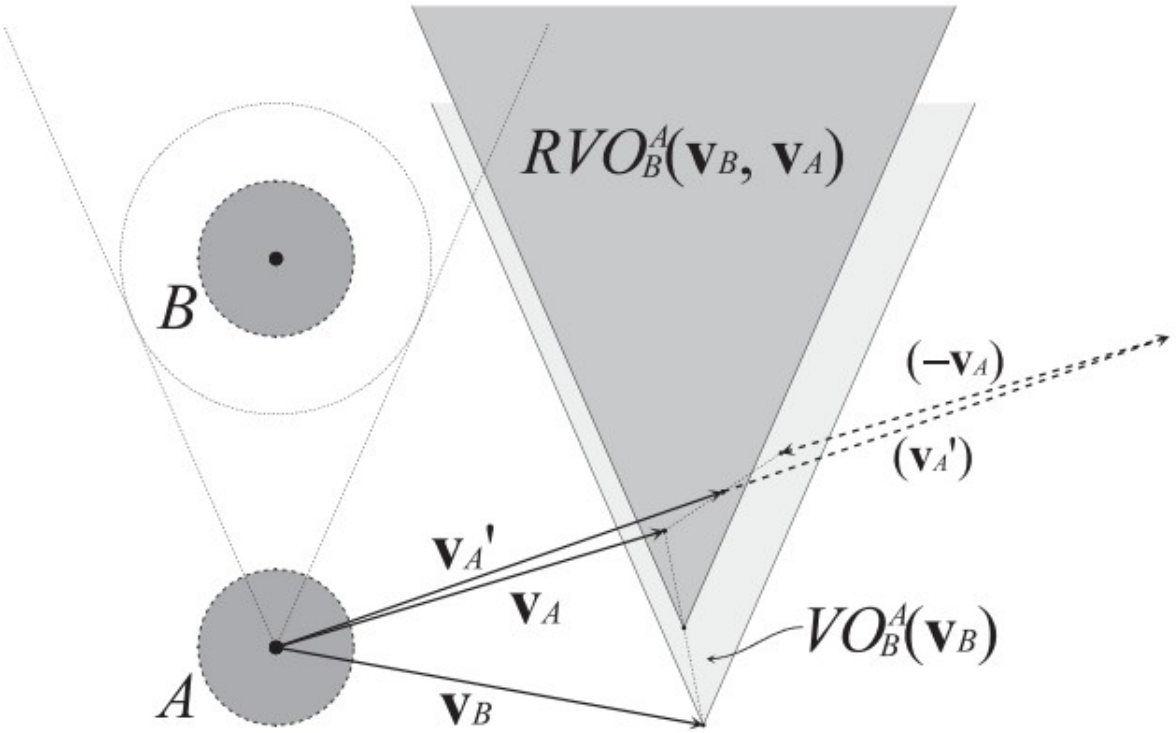


Figura 7.5: RVO de un obstáculo B respecto un agente A

7.4 Configuración de los parámetros

Un problema muy común en la inteligencia artificial es la configuración de los sistemas. Una vez el sistema está implementado, sólo falta elegir el valor de todos aquellos parámetros que distinguen a un sistema mediocre de un sistema perfecto e inteligente, aunque la implementación sea la misma.

En nuestro caso no ha sido diferente dada la elevada cantidad de parámetros por configurar. Especialmente ha sido difícil ponderar los diferentes tests de las consultas *EQS.* Es decir, determinar la importancia de cada factor al elegir la posición de ataque, como por ejemplo la importancia de la distancia al jugador, al agente o a una cobertura a la hora de elegir una posición de ataque. No obstante, también ha habido otros parámetros difíciles de configurar, como por ejemplo la velocidad de desplazamiento, el daño aceptable antes de cubrirse, o simplemente cuantas balas tiene que haber gastado un agente para considerar que debe recargar. En general, no hay una solución universal y exacta. La única manera de hacerlo es empíricamente.

A medida que el desarrollo del proyecto fue avanzando, hemos ido modificando los parámetros y haciendo muchas pruebas para verificar que el comportamiento de los *NPCs* era correcto. Gracias a nuestros conocimientos como jugadores frecuentes de este tipo de juegos, no partimos de cero, si no que *a priori* ya teníamos una idea de que valores aproximados podían ajustarse mejor para cada parámetro y por lo tanto,

tener una base de donde partir.

7.5 Depuración

Sin duda alguna, una de las partes más complicadas de todo el proyecto ha sido la depuración del sistema de inteligencia artificial en general. Aunque su implementación podía ser más o menos complicada, comprobar su funcionamiento resultaba muy complejo.

Concretamente, implementamos un sistema para asegurarnos de que tanto el algoritmo de visibilidad como el mapa de influencia funcionaban correctamente. Aunque su implementación no fue complicada desde un punto de vista conceptual, si lo fue desde un punto de visa técnico ya que tuvimos que implementar funciones a un nivel bastante bajo para modificar imágenes y texturas.

Dado que tanto la propagación de la influencia como la visibilidad operan siempre en posiciones de la *Navmesh* decidimos crear un sistema de depuración que consistia en reutilizar la imagen de la *Navmesh* generada en el apartado *6.4.4.1.* Esta imagen con la misma forma del mapa desde una perspectiva superior la aplicamos como textura al suelo del mapa y por último creamos una función que nos permitía modificar el color de una posición y un conjunto de funciones que nos permitían convertir una posición de espacio real a espacio de textura y viceversa (ver Figura 7.6). De esta manera, cualquier punto del mapa lo podemos representar en la imagen 2D y resulta mucho más fácil depurar el funcionamiento de los algoritmos.

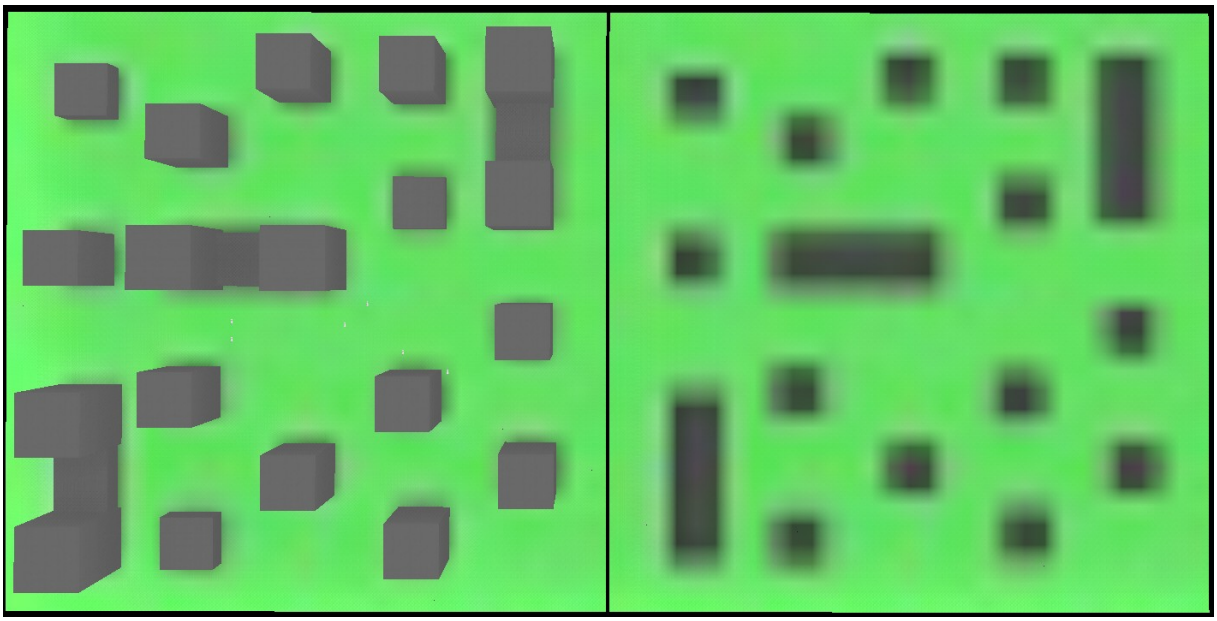


Figura 7.6: A la izquierda el mapa original y a la derecha la textura de baja resolución (64x64) utilizada para la depuración

Por ejemplo, para depurar el mapa de influencia (ver Figura 7.7) modificamos el componente R de color RGB de acuerdo al valor de influencia de aquella posición. De esta manera resulta mucho más fácil ver como se propaga la influencia. De hecho, podemos realizar visualizaciones en tiempo real (ver vídeos en el apartado Error: Reference source not found y Error: Reference source not found).

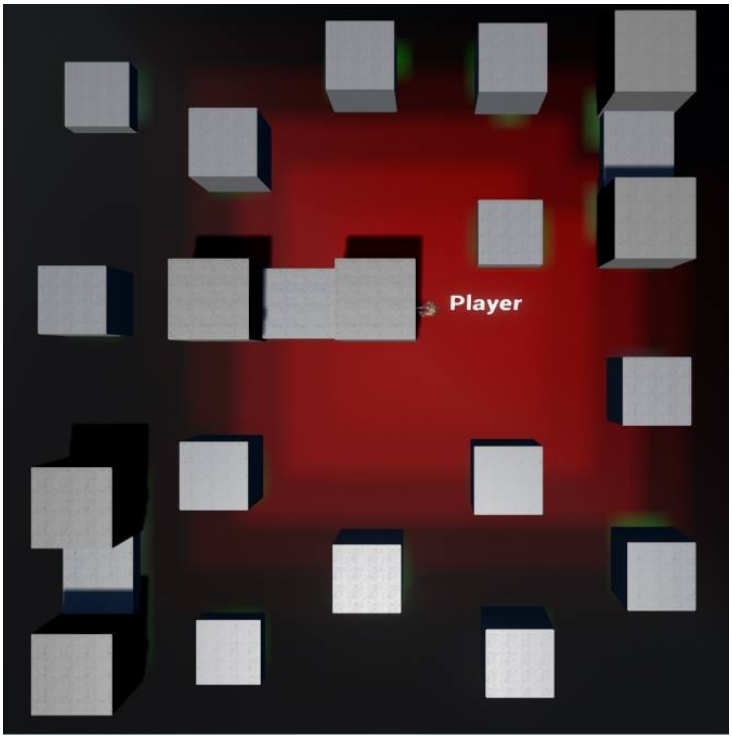


Figura 7.7: Propagación de la influencia alrededor del jugador

7.6 Situaciones de supresión

Implementar la tarea de supresión no fue un problema complejo. En primer lugar diseñamos la consulta *EQS* de supresión (ver apartado 5.4.2.4.6) y después implementamos la tarea (ver apartado 6.5) encargada de disparar a aquellas posiciones.

El gran problema fue decidir cuándo el agente debería aplicar fuego de supresión. Es decir, en qué árbol de comportamiento y en qué condiciones concretas el *NPC* debería suprimir al jugador.

A través del estudio y razonamiento del comportamiento humano, intentamos entender en que situaciones de combate aplicaríamos fuego de supresión y llegamos a las siguientes conclusiones:

- Es muy importante que sepamos de manera muy clara la posición en la que se encuentra el jugador al que vamos a aplicarle fuego de supresión. Si no

conocemos con certeza su posición, no tiene sentido gastar munición y desvelar nuestra posición.

- Unicamente vale la pena aplicar supresión cuando los agentes están modificando sus posiciones de ataque para situarse cerca del jugador. De esta manera, al mantener al personaje acorralado, los *NPCs* pueden desplazarse de manera segura.
- Si el jugador es visible por algún agente, no vale la pena aplicar fuego de supresión.
- No tiene sentido que diversos agentes apliquen supresión a un mismo punto. Deberían dividirse las posiciones e intentar contener al jugador por todos los lados.

Una vez definidas las condiciones y los requisitos que definían las situaciones concretas en las que debería aplicarse el comportamiento de supresión, intentamos sintetizarlas e incluirlas en nuestro sistema. No obstante, esto también resultó complicado, por lo que tuvimos que simplificar un poco las condiciones y el comportamiento.

Finalmente, decidimos que activaríamos la tarea de supresión cuando el jugador no fuese visible (**!***PlayerIsVisible)* pero supiésemos donde esta con una alta probabilidad (**!***PlayerIsLost*). En ese momento, los agentes calculan posiciones de supresión y disparan a la misma vez que avanzan, procurando distribuirse de manera uniforme por los diferentes flancos (ver apartado 5.2.5.5.2).

7.7 Problemas varios

Muchos de los inconvenientes a los que nos hemos enfrentado estaban íntimamente relacionados con el motor de *Unreal*. Por ejemplo, uno de los problemas que más costó solucionar fue modificar la heurística del *A*.* Encontrar las funciones y clases que se debían modificar para cambiar el *A** llevó semanas y no lo hubiésemos conseguido sin la ayuda de la comunidad de *UE4 [42].*También nos encontramos con algunos problemas para definir los tests personalizados de las consultas *EQS,* así como para implementar la estructura de datos basado en una imagen para la depuración.

No obstante, también nos encontramos con problemas que no estaban relacionados con la implementación. Estos problemas tenían una base más conceptual y correspondían a un fallo de diseño. Prácticamente todos ellos los hemos comentando a lo largo de la memoria, como por ejemplo el fuego amigo, el cálculo de la visibilidad o cómo predecir la posición del jugador.

08

Resultados

8 Resultados

8.1 Introducción

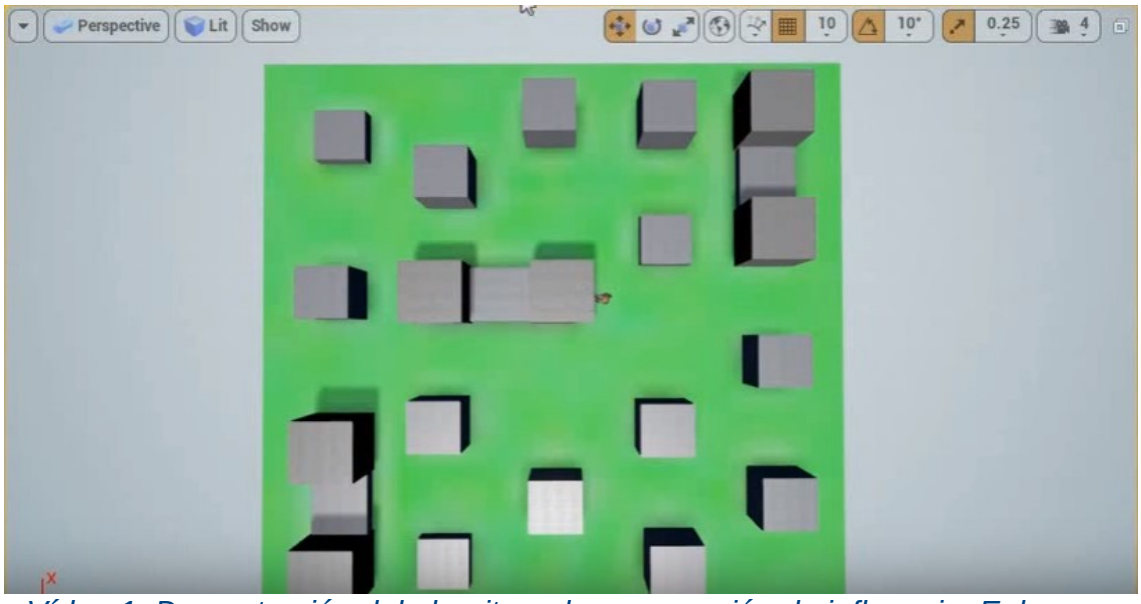
Finalizado el diseño e implementación del sistema, comprobaremos su funcionamiento y el resultado de meses de trabajo. La mejor manera de probar este sistema es jugar contra el. Como esto no es posible en un documento escrito, nos ayudaremos de videos para mostrar el funcionamiento de la IA, ya que consideramos que es el formato más parecido.

A continuación veremos el funcionamiento de algunos algoritmos que han tenido una relevancia importante en el proyecto así como el comportamiento de la IA en diferentes situaciones de combate.

8.2 Propagación de la influencia

En el vídeo 1 podemos ver cómo funciona el algoritmo de propagación de la influencia y su depuración.

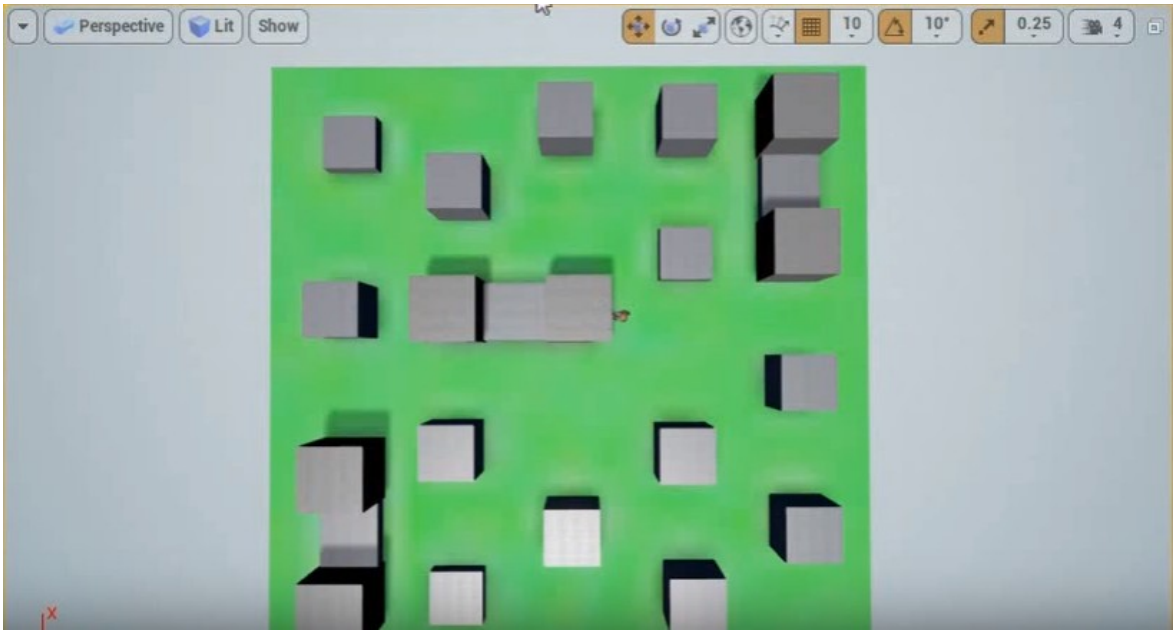
En el “suelo” del mapa esta aplicada la imagen para la depuración. Entonces, a partir de la posición inicial del jugador, la influencia se propaga en todas las direcciones por aquellas posiciones que son navegables. En la textura del suelo podemos ver como la componente R de cada píxel cambia a medida que la influencia se propaga a través del mapa, asignando un valor de R alto a aquellas posiciones con alta influencia y una R baja a las posiciones con baja influencia.



Vídeo 1: Demostración del algoritmo de propagación de influencia. Enlace http://youtu.be/VE70oB4xyrw

8.3 Propagación de la influencia y visibilidad

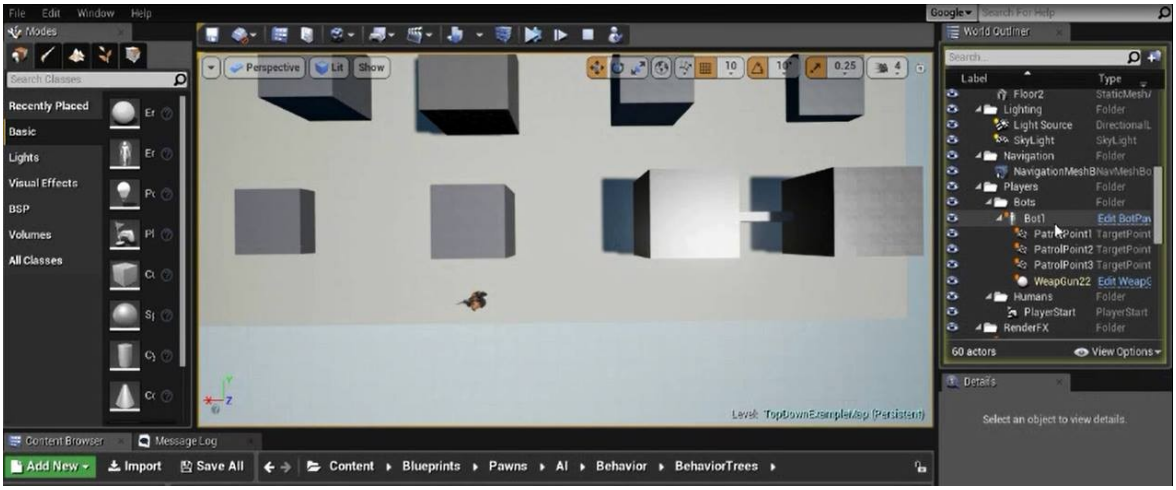
En el vídeo 2 vemos como funciona el algoritmo de propagación de influencia pero teniendo en cuenta la visibilidad del *NPC*. A diferencia de antes, la influencia no se propaga por todas las posiciones navegables si no solo por aquellas que no son visibles por el *NPC.* Esto tiene sentido ya que si el *NPC* ve una determinada zona, tiene la certeza de que el jugador no podrá pasar por ahí sin que el lo vea. Por lo tanto no tiene sentido propagar la influencia por esa área. No obstante, cuando el agente pierde la visibilidad de una zona, esta tarda vuelve a propagar la influencia, aunque tarda unos segundos en poder hacerlo de manera completa. Esto lo hemos hecho siguiendo el razonamiento de que vale más la pena visitar aquellas zonas que no hemos visitado que las que ya hemos visto.



Vídeo 2: Demostración del algoritmo de propagación de influencia teniendo en cuenta la visibilidad de un NPC. Enlace http://youtu.be/j0cO867Jg7A

8.4 Comportamiento Patrol

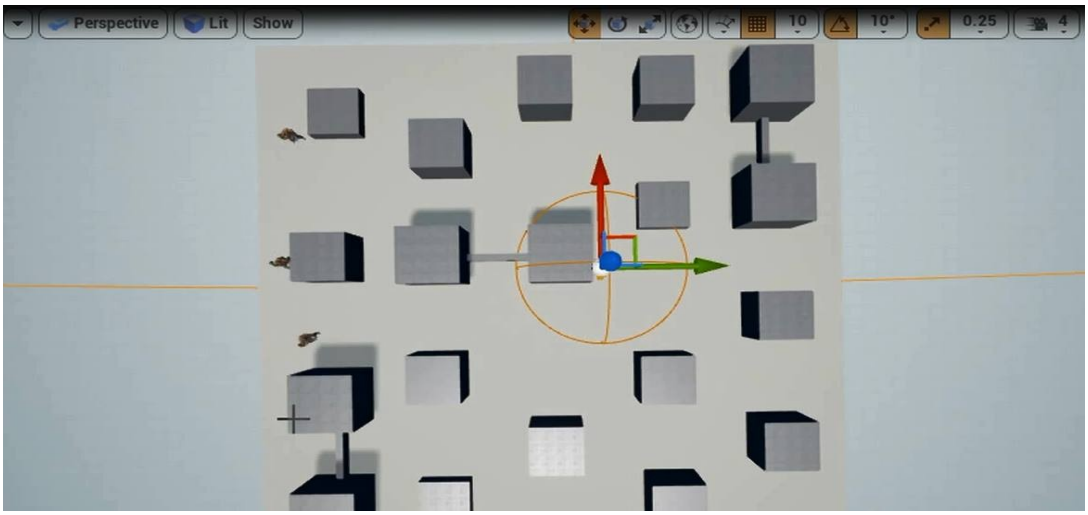
En el siguiente vídeo 3 podemos ver como funciona el comportamiento de patrulla. En él vemos a un *NPC* con un conjunto de puntos definidos como puntos de patrulla. El comportamiento de patrulla (explicado anteriormente en el apartado 5.2.3) consiste en moverse de un punto a otro esperando un tiempo aleatorio al llegar a cada punto. Al final del video, podemos ver la ejecución en tiempo real del árbol de comportamiento.



Vídeo 3: Demostración del comportamiento de patrulla. Enlace http://youtu.be/GxMbbNWw-DA

8.5 *Comportamiento Search*

En el siguiente Vídeo 4 podemos ver como cuatro agentes buscan al jugador después de que éste haya delatado su posición mediante un ruido. Vemos como, para encontrar al jugador, estos agentes se distribuyen y se desplazan desde diferentes ángulos a la posición desde la cual se origino el ruido. Al llegar a esa posición los agentes no encuentran al jugador porque se ha escondido. En ese momento los agentes re-evaluan la situación y se re-distribuyen por las posiciones más probables en las que podría estar. Poco tiempo después un agente encuentra al jugador y notifica a los demás. En ese mismo instante el comportamiento de búsqueda acaba y comienza el comportamiento de ataque.



Vídeo 4: Demostración del comportamiento de búsqueda. Enlace http://youtu.be/fTocYPT-k6o

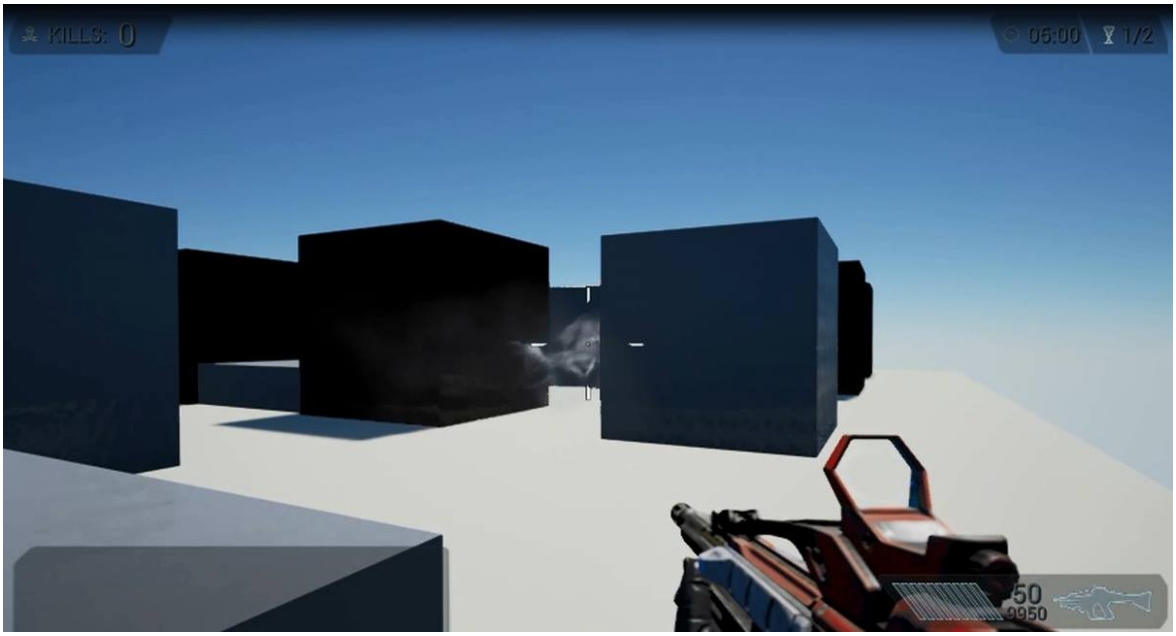
Vale la pena destacar que tras hacer diversas pruebas con cuatro agentes, resulta

prácticamente imposible permanecer más de 5 o 10 segundos sin que los agentes encuentren al jugador. Se distribuyen de manera tan correcta y cubren tanta superficie en tan poco tiempo que resulta muy complicado evitarlos, incluso desde una perspectiva superior y sabiendo donde se encuentran en todo momento.

8.6 Comportamiento Fight

8.6.1 Rama de cobertura

En el vídeo 5 podemos ver como el agente inteligente busca y se desplaza hacia una posición segura a resguardo del jugador. En él, el *NPC* utiliza un comportamiento de cobertura simplificado que hace que el agente se cubra tan pronto como vea al jugador. De esta manera es más fácil centrarnos en el comportamiento que nos importa.



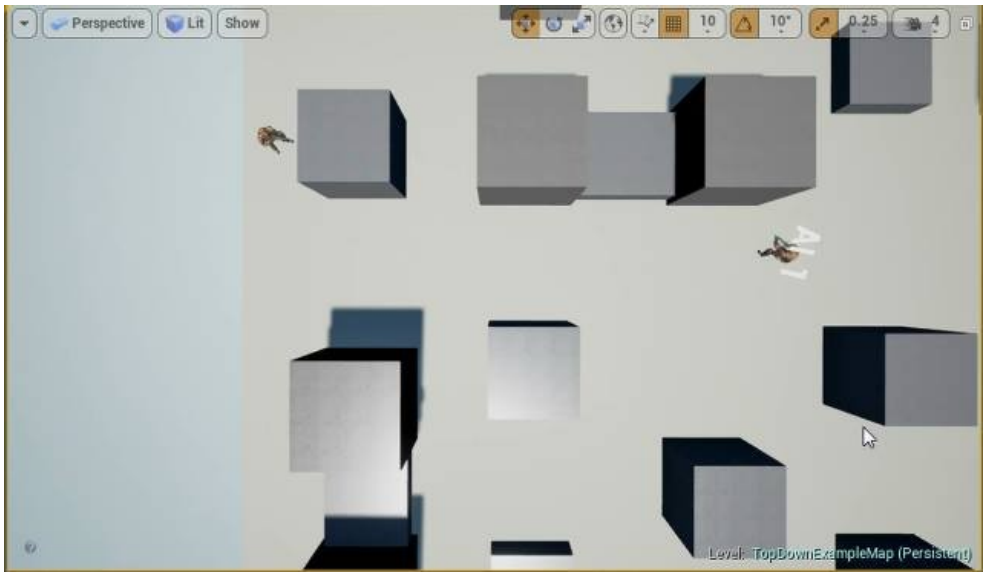
Vídeo 5: Demostración del comportamiento de cobertura final. Enlace http://youtu.be/EW3r7214_X0

En cambio, en el vídeo 6 se ejecuta el comportamiento de cobertura explicado en la memoria de manera que el agente sólo se cubre si necesita recargar o si está recibiendo un daño significativo. Como podemos ver, en cuanto el agente se queda sin munición busca un lugar seguro para recargar. De la misma manera, cuando el agente sufre daño también busca una cobertura segura para esconderse.

Vídeo 6: Demostración del comportamiento de cobertura simplificado. Enlace http://youtu.be/gyrZ40zJT3I

8.6.2 Rama de avance

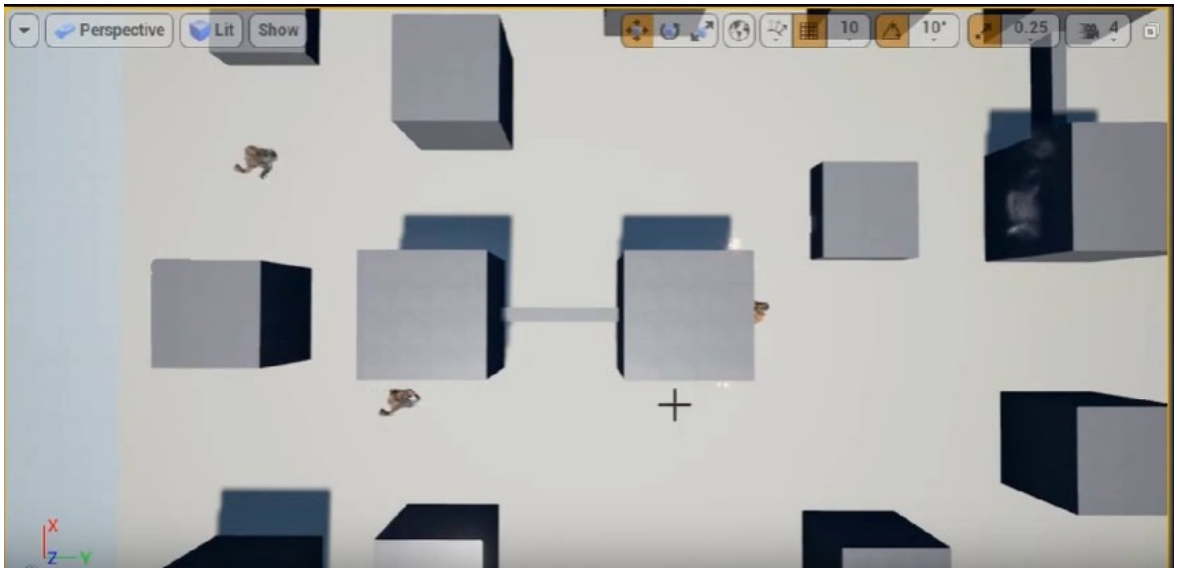
En el vídeo 7 el agente utiliza un comportamiento formado únicamente por la rama de avance. Esta rama sólo se activa cuando el jugador no es visible, ya que el *NPC* aprovecha esta situación para acercarse a él. Como podemos ver, al ejecutarse solamente el comportamiento de avance, el *NPC* permanece quieto mientras el jugador es visible y en cuanto deja de serlo, el comportamiento se activa y el *NPC* se prepara, recarga y avanza, buscando una posición de ataque más cercana al jugador para presionarlo.



Vídeo 7: Demostración del comportamiento de avance. Enlace http://youtu.be/ibkQxci0enk

8.6.2.1 *Supresión*

En el vídeo 8 podemos ver como dos *NPC* ejecutar la rama de comportamiento correspondiente a la supresión. Los agentes han oído al jugador y saben que se encuentra detrás de este obstáculo. Por lo tanto, deciden avanzar hacia la posición del jugador a la vez que aplican fuego de supresión para mantener al jugador acorralado. Cada uno de los *NPCs* avanzan por un lado diferente y aplican fuego de supresión en posiciones diferentes impidiendo al jugador cualquier posibilidad de atacar.

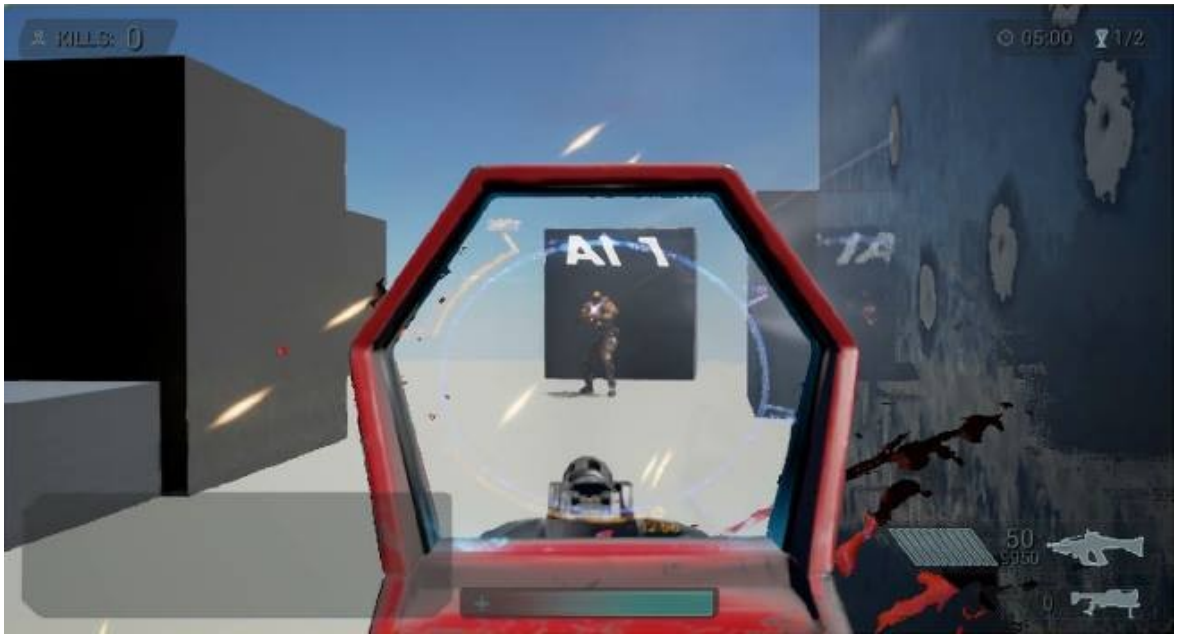


Vídeo 8: Demostración del comportamiento de supresión. Enlace http://youtu.be/JWuTSY9xDPg

8.6.3 Rama de ataque

El vídeo 9 un agente inteligente está ejecutando el comportamiento de ataque. En cuanto éste agente ve al jugador lo comienza a disparar. El agente permanece quieto en su posición disparando ya que considera que es segura. En cuanto el agente recibe daño, esta posicion deja de ser segura y por lo tanto decide buscar una nueva posición de ataque para disparar al jugador. Por último, cuando la distancia entre el agente y el jugador es muy pequeña, el agente se dirige directamente hacia la posición del jugador para atacarlo.

Vale la pena destacar la similitud de este comportamiento con el comportamiento de avance en el sentido de que ambos buscan posiciones de ataque, aunque cada uno lo hace en circumstancias diferentes.



Vídeo 9: Demostración del comportamiento de ataque. Enlace http://youtu.be/wSTioum0eas

8.6.3.1 *Ataque en grupo*

En el vídeo 10 los agentes inteligentes llevan a cabo un ataque en grupo desde diferentes flancos. Cada uno de ellos busca una posición de ataque de tal manera que el jugador acabe rodeado y sea muy difícil repeler el ataque.

Vídeo 10: Demostración de un ataque en grupo. Enlace http://youtu.be/zs1O1GdHL7U

8.7 *Gameplay*

Una vez demostrados, en menor o mayor medida los diferentes comportamientos de los agentes de manera separada, veremos su combinación en único y gran árbol. Los comportamientos que antes podían parecer pobres, al combinarlos en un sólo sistema consiguen crear una verdadera ilusión de inteligencia.

Primero analizaremos una situación en la que un jugador lucha contra un único agente y luego un jugador contra cuatro agentes.

8.7.1 Un jugador contra un agente

El vídeo 11 es una partida completa en la que un único jugador se enfrenta a un único agente el cual utiliza el árbol de comportamiento completo, realizando operaciones de patrulla, búsqueda, defensa, ataque, avance y supresión. A ambos personajes se les ha incrementado significativamente la vida con el objetivo de alargar el combate durante más tiempo y permitir que todos los comportamientos se activen.

El agente comienza en el comportamiento de patrulla. Cuando el jugador lo ve y le dispara, activa automaticamente el comportamiento de combate y el agente empieza a disparar. Tras sufrir una cantidad considerable de daño, el comportamiento de covertura se activa, el agente se esconde y aprovecha para recargar. Acto seguido, el jugador decide hacer lo mismo. En ese momento, aprovechando que el jugador esta recargando, se activa el comportamiento de avance y el agente se acerca hacia el jugador. Cuando el jugador acaba de recargar y vuelve a atacar al agente este decide cambiar de posición ya que habia quedado muy expuesto y estaba sufriendo mucho daño. A continuación, el jugador decide cambiar de posición para intentar sorprender al agente. Sin embargo, no consiguio sorprender al agente el cual ya estaba preparado para luchar. Tras intercambiar unos disparos el agente vuelve a su cobertura y el jugador busca una nueva posición de ataque. En ese momento, el agente también decide cambiar su posición de ataque (pasa del lado derecho a izquierdo) para sorprender al jugador. Por último, en un intento de flanquear al agente, el jugador decide cambiar de posición de ataque. No obstante y sin apenas darse cuenta, el jugador ha sido flanqueado por el agente que se ha situado a su espalda y le comienza a disparar. Por último, dado que el agente estaba gravemente herido, muere.

Hemos podido ver en este video que la situación de combate ha sido muy emocionante. Todo ha sucedido rápidamente y la situación se ha desarrollado de manera dinamica y poco predecible. El agente fue capaz de cubrirse cuando fue necesario y sacar el maximo beneficio de la situación para avanzar, atacar o flanquear. Esto ha resultado en una experiencia de juego muy satisfactoria ya que, aunque el agente murió, lucho de una manera inteligente, provocando que el jugador tuviese que esforzarse para ganar, generando una gran satisfacción tras la victoria.

Vídeo 11: Demostración de un jugador luchando contra un agente. Enlace http://youtu.be/sQ0by2-ijB8

8.7.2 *Un jugador contra cuatro agentes*

A diferencia de los demás vídeos, esta vez hemos eliminado la voz para que se puediese oir el audio del juego. Consideramos que en esta ocasión si es importante para que la experiencia de juego sea completa y podamos sentir las mismas sensaciones que el jugador: escuchar los pasos cuando se acercan los enemigos, sentir el fuego de supresión, oir las armas recargar, etc.

En este último vídeo 12 he unido diferentes partidas en las cuales un jugador se enfrenta a cuatro agentes que combaten de manera conjunta. En él podemos ver como los agentes llevan a cabo los diferentes comportamientos de patrulla, búsqueda y combate con sus respectivas ramas de defensa, avance, ataque y supresión.

Creemos que la similaridad entre las diferentes partidas incluidas en el vídeo y por tanto, el rasgo que caracteriza al sistema de IA, es la capacidad de presionar y de generar una cierta ansiedad sobre el jugador como si de una batalla real se tratara. Este sentimiento es originado por el fuego de supresión que impide que el jugador se mueva libremente; por la efectividad en las operaciones de búsqueda que no permiten al jugador descansar de la batalla; y por el ataque desde multiples flancos. Además, el hecho de incluir cuatro agentes en vez de uno en la partida hace mucho más complicado para el jugador predecir las posiciones de los *NPCs*. Esto, sumado al hecho de que el jugador esta en inferioridad númerica, provoca que el jugador tenga que moverse muy rápido y cubrir multiples flancos para evitar ser sorprendido.

Todo esto resulta en una sensación final, que independientemente de la dificultad del juego, es la de una verdadera guerra. Todo sucede de manera muy rápida e

impredecible. El jugador se encuentra solo contra cuatro enemigos y no tiene tiempo para pensar. La única manera de sobrevivir es ser más rápido y, sobretodo, más inteligente que los enemigos. Una vez el jugador lo consigue, estos sentimientos negativos de estres y ansiedad desaparecen y son substituidos por sentimientos de grandeza y satisfacción por haber sido capaz de ganar la batalla pese a haber luchado solo.

Vídeo 12: Demostración de un jugador luchando contra cuatro agentes. Enlace http://youtu.be/TgmPYbJQutw

09

Conclusiones

9 Conclusiones

9.1 Introducción

Al iniciar este proyecto, apenas conocíamos el funcionamiento o la importancia de la IA en los videojuegos. Tras meses de investigación, diseño y desarrollo hemos conseguido ampliar nuestra comprensión de la IA así cómo de las técnicas más utilizadas para modelar su comportamiento en los juegos de tipo *FPS*.

Este proyecto no es sólo la culminación de meses de duro trabajo si no también una muestra inequívoca de la consolidación de los conocimientos adquiridos durante su desarrollo.

Como resultado hemos obtenido un sistema de IA que simula satisfactoriamente el comportamiento humano en situaciones de combate llegando a crear momentos de verdadero estres y ansiedad para el jugador y finalizando con una gran satisfacción. Todo esto contribuye a una experiencia de juego muy adecuada para este género de *FPS.*

A continuación evaluaremos el resultado del proyecto así como su desarrollo y finalizaremos con unas breves reflexiones sobre el proyecto.

9.2 Evaluación del resultado

Para evaluar si el resultado del proyecto ha sido realmente satisfactorio evaluaremos el grado de cumplimiento de cada uno de los objetivos que planteamos al inicio del proyecto (ver apartado 1.2) y también describiremos brevemente el resultado del proyecto.

9.2.1 Cumplimiento de los objetivos

9.2.1.1 Tactical pathfinding

Mediante la modificación del algoritmo de *pathfinding* de *Unreal* para incluir la seguridad de un camino,¡ como heurística junto con la distancia, hemos conseguido que los agentes inteligentes se desplacen de manera correcta desde un punto de vista táctico. De esta manera, a la hora de buscar un camino, el sistema tiene en cuenta la visibilidad del jugador humano para evitar las zonas visibles y utilizar caminos seguros.

Por lo tanto, podemos concluir que este objetivo se ha cumplido satisfactoriamente y que gracias a el los agentes inteligentes parecen más humanos ya que evitan comportamientos “estúpidos” como por ejemplo cruzar por zonas peligrosas. No obstante, como comentamos en el trabajo futuro (apartado 10.5), estos agentes no evitan cruzar las lineas de fuego amigo.

9.2.1.2 *Razonamiento espacial dinámico*

Recordemos que este objetivo hace referencia a la capacidad de los agentes de detectar los cambios del entorno de manera rápida y actuar consecuentemente. Gracias al sistema de percepción de *Unreal,* la implementación de diversos algoritmos y los árboles de comportamiento hemos sido capaces de replicar todo este proceso. Desde la detección de los cambios hasta la ejecución de las acciones pertinentes obtenidas a través del proceso de toma de decisiones. De esta manera el sistema es capaz de analizar el entorno y predecir la posición del jugador, las mejores posiciones para atacarlo o cubrirnos e incluso a que zonas aplicar fuego de supresión. Todo ello en tiempo real y dependiendo de diversos factores cambiantes como la última posición del jugador, mi posición, la posición de los otros agentes o la distribución del entorno.

Por estos motivos consideramos que hemos cumplido el segundo objetivo de razonamiento espacial dinámico. Sin embargo, nos hubiese gustado utilizar un escenario dinámico para enriquecer la experiencia de juego (ver apartado 10.6)

9.2.1.3 *Predicción del jugador*

El tercer objetivo era dotar al sistema de IA de una herramienta que le permitiese simular el pensamiento humano para predecir la posición del jugador durante los comportamientos de búsqueda. En nuestro caso la técnica que hemos utilizado esta basada en mapas de influencia, donde el valor de la influencia de una posición esta directamente relacionado con la probabilidad que tiene el jugador de encontrarse en aquella posición. A medida que el tiempo pasa la probabilidad se propaga generando múltiples posiciones probables en las cuales podría estar el jugador.

Con esta idea tan simple hemos conseguido crear un algoritmo de predicción extremadamente simple y efectivo. Los agentes son capaces de encontrar al jugador en pocos segundos tras ejecutar este sistema. Estamos sumamente satisfechos con el resultado de este sistema y damos por cumplido este objetivo.

9.2.1.4 *Coordinación colectiva*

Otro de los objetivos más importantes del proyecto era conseguir una coordinación y cooperación entre los diferentes agentes para llevar a cabo acciones complejas.

Cabe destacar que para poder implementar la coordinación colectiva primero debíamos finalizar todos los otros aspectos del juego. Por esta razón no hemos tenido tiempo suficiente para implementar un sistema de IA grupal.

No obstante, hemos sido capaces de implementar a los agentes inteligentes para que tuviesen en cuenta cierta información grupal a la hora de tomar ciertas decisiones, como por ejemplo, la posición de los otros agentes o sus posiciones de supresión, etc. De esta manera, pese a no existir explícitamente un sistema de IA colectiva, hemos conseguido generar una ilusión de cooperación entre los agentes

que emerge unicamente de su comportamiento individual.

Personalmente, consideramos que el objetivo se cumple si el jugador se sumerge en esta ilusión de cooperación y percibe a todos los agentes como un único enemigo trabajando conjuntamente para acabar con el.

Por este motivo, aunque no hayamos implementando un sistema de IA colectiva como hubiésemos deseado (ver apartado 10.1), creemos que el objetivo se cumple. Al menos siempre que consigamos crear esta ilusión de cooperación.

9.2.1.5 Rendimiento

El último objetivo que nos habíamos planteado era mantener un nivel alto de rendimiento de manera que la experiencia del juego no se viese afectada por los cálculos necesarios por el sistema de IA.

Aunque aún hay cierto margen de mejora en algunos aspectos relacionados con el rendimiento (ver apartado 10.9), estamos muy contentos con el resultado ya que en ningún caso afecta a la experiencia de juego. El diseño y la implementación de los algoritmos así como de las estructuras de datos que utilizamos siempre se han llevado a cabo teniendo en cuenta el tiempo de ejecución y la memoria necesaria.

9.2.2 Descripción del resultado

Aunque la mejor manera de describir el resultado es mediante vídeos (ver apartado 8) consideramos relevante describir brevemente el sistema de IA.

Este sistema, implementado sobre *Unreal Engine 4*, esta basado en *Behavior Trees* y su objetivo es simular el comportamiento de los agentes inteligentes en un *FPS.* Aunque hemos implementado diversos comportamientos nos hemos centrado en las situaciones de combate.

En estas situaciones, los agentes inteligentes son capaces de cubrirse para recargar o simplemente para protegerse. También son capaces de encontrar posiciones seguras desde las cuales atacar, así como de cambiar su posición de cobertura o ataque de manera frecuente, tal y como lo haría un jugador real. Por otro lado, pueden desarrollar operaciones de avance para acercarse al jugador y presionarlo mientras le aplican fuego de supresión.

Lo verdaderamente importante de este sistema, que solo se puede comprobar cuando se juega contra el, son los sentimientos que es capaz de generar en el jugador. Estos sentimientos se corresponden, en un principio, con el estres y la ansiedad provocados por una ofensiva contínua y desde multiples posiciones escogidas estrategicamente. Después, cuando el jugador es capaz de sumergirse en la experiencia de juego y controlarla surge una nueva sensación de grandeza que le permite al jugador acabar con todos los *NPCs* y sentir un sentimiento de superación y satisfacción.

9.3 Desarrollo

Aunque obviamente el resultado del proyecto es fundamental, vale la pena destacar la importancia del proceso de aprendizaje que se ha llevado a cabo y que nos ha permitido conseguir este resultado.

A continuación detallaremos los aspectos más relevantes de este proceso relacionados con la inteligencia y los videojuegos y con *UE4*, el motor utilizado para desarrollar el proyecto.

9.3.1 Inteligencia artificial y videojuegos

Antes de comenzar la implementación del sistema, investigamos durante bastante tiempo los sistemas más populares utilizados en la IA aplicada a videojuegos. Aunque si encontramos una cantidad considerable de artículos y presentaciones, en muchas ocasiones solo proveían explicaciones conceptuales a un nivel muy alto que no servían de mucho. No obstante, algunas de estas empresas iban un poco más allá y proporcionan información realmente útil. Vale la pena destacar dos empresas que han tenido una gran influencia en este proyecto: la empresa *Guerilla,* creadores de la serie *Killzone;* y a *AigameDev.com,* el repositorio *online* más gran de inteligencia artificial en videojuegos.

Gracias a esta investigación, fuimos capaces de entender el funcionamiento, las ventajas e inconvenientes de las técnicas más populares para modelar la inteligencia artificial en los diferentes géneros de videojuegos. Pese a únicamente haber profundizado en la técnica basada en los árboles de comportamiento, ya que la utilizada en este proyecto, consideramos que es vital haber investigado las otras técnicas. Es muy importante que los desarrolladores de IA conozcan las diferentes maneras de atacar un problema para poder hacerlo de la manera más adecuada dependiendo de las circunstancias.

Una de las técnicas que más nos llamo la atención durante el desarrollo del proyecto fueron los mapas de influencia. Su elevada simplicidad y potencia, sin duda alguna lo hacen una herramienta imprescindible que todo desarrollador de IA debería conocer junto con técnicas de *pathfinding* y algoritmos para realizar el cálculo de la visibilidad.

Por otro lado, nos resultaron especialmente interesantes los sistemas basados en planificadores como los *GOAP* o las *HTN*. Su funcionamiento nos parece realmente curioso y se asemeja enormemente al pensamiento humano. Dadas las restricciones de tiempo en este proyecto, decidimos utilizar árboles de comportamiento. No obstante, de haber contado con más tiempo nos hubiese gustado implementar un sistema basado en redes jerárquicas de tareas.

Vale la pena destacar la dificultad de depurar estos sistemas de IA. Aunque *UE4* incluye herramientas muy útiles para la depuración, dada la naturaleza dinamica de

estas situaciones resulta muy dificil poder replicarlas. Cualquier error es muy dificil de reproducir, y más aun si hay multiples agentes en la partida. Arreglar, pero sobretodo, detectar cualquier fallo requiere muchas horas de juego para ser capaz de averiguar las condiciones exactas en las que surge el problema.

9.3.2 *Unreal Engine 4*

Usar *Unreal Engine 4* como el motor para el desarrollo del juego fue una decisión muy acertada y que volveríamos a tomar. No obstante hay algunos aspectos de *Unreal* que vale la pena destacar.

Por un lado, las herramientas de IA que incorpora son fascinantes y muy potentes. Nos han facilitado mucho el trabajo y nos han permitido implementar características que en otros sistemas hubiesen sido imposibles.

Sin embargo, tal y como preveíamos, la curva de aprendizaje de *Unreal* es muy pronunciada. El proceso de aprendizaje es francamente lento. La documentación, sobretodo en temas de IA, es totalmente inexistente. En la mayoría de casos, hemos tenido que mirar el código fuente del motor (en C++) directamente para entender el funcionamiento de algunos algoritmos y saber las modificaciones pertinentes que había que hacer. Todo esto resulto en una progresión muy lenta del proyecto y, sinceramente, en mucha frustración. Durante los dos o tres primeros meses el proyecto fue muy lento e incluso llego a estar atascado durante algunas semanas. No fue hasta el tercer o cuarto mes cuando el proyecto comenzó a tomar forma y empezamos a ganar fluidez con el motor y con la implementación en C++. A partir de ese momento, fuimos capaces de implementar nuevas ideas en cuestión de días, por lo que pudimos recuperar mucho del atraso acumulado en pocas semanas.

Sinceramente, creemos que *Unreal* es un motor increíble y con un gran potencial. Los resultados que se pueden obtener con el son mucho mejores que los que se podrían obtener con un motor más simple y sencillo. Por esta razón, es lógico que su curva de aprendizaje sea más pronunciada. Cabe destacar que si se decide utilizarlo, hay que estar mentalmente preparado para una progresión muy lenta en los primeros meses de trabajo.

9.3.3 *Experiencia personal*

Desde un punto de vista personal este proyecto nos ha ayudado a consolidarnos como personas y como ingenieros informáticos.

Desarrollar este proyecto nos ha enseñado a seguir adelante a pesar de las adversidades. Aunque no tuviésemos conocimientos previos de IA en videojuegos y no supiésemos por donde empezar decidimos realizar este trabajo. Pese a la falta de documentación y dificultades con la implementación no nos rendimos y seguimos trabajando para conseguir los objetivos propuestos.

Tras muchas horas de trabajo, algunas decepciones y grandes momentos, conseguimos, partiendo de un sistema de inteligencia artificial prácticamente nulo, un resultado muy satisfactorio: un sistema artificial que simula el comportamiento de los humanos en situaciones de combate.

Con toda seguridad podemos decir que el sentimiento que ha generado estos grandes momentos y que ha permitido superar las decepciones es aquel que surge cuando consigues desgranar tu razonamiento personal a la hora de jugar y eres capaz de incorporarlo al sistema. Es muy divertido y a la vez muy grato ver como los agentes se comportan tal y como tú lo harías.

Si este proyecto ha sido satisfactorio es, sin ninguna duda, gracias a la cantidad de horas que invertimos en busca de este sentimiento que ha conducido el desarrollo del proyecto.

9.4 *Punto final*

Aunque indudablemente hay algunos aspectos del proyecto potencialmente mejorables (ver apartado 10) estamos muy contentos, tanto por el resultado como por todo lo que hemos aprendido durante su desarrollo, desde técnicas para modelar comportamientos de inteligencia artificial hasta su implementación en C++. Vale la pena destacar que este proyecto no ha sido realmente difícil de implementar (a excepción de algunos algoritmos). La verdadera dificultad esta en encontrar la manera de transmitir y dotar a los agentes de un comportamiento humano. Se trata de pensar más que de programar.

No obstante, ha sido, en general, un trabajo difícil y debemos admitir que *a priori* nos daba miedo de llevar a cabo un proyecto de esta índole. Pero tras muchas dudas, muchas horas de trabajo y sobretodo mucho esfuerzo, hemos conseguido acabarlo de manera muy satisfactoria.

Por otro lado, ha sido muy divertido trabajar desde un punto de vista más interno un género de videojuegos que siempre nos ha fascinado. Sin duda alguna, la mejor parte ha sido utilizar los conocimientos como jugador para incluirlos en el sistema. A partir de ahora no volveremos a ver los *FPS* con los ojos de un jugador, si no con los de un desarrollador de IA. Siempre intentando entender el proceso de toma de decisiones, la información que lo conduce y como afecta cada accion al entorno.

10

Trabajo futuro

10 Trabajo futuro

10.1 Comportamiento grupal

Tal y como hemos comentado anteriormente, este ha sido el aspecto más complejo del proyecto y al que menos tiempo le he podido dedicar ya que cronológicamente se encuentra al final.

Nos hubiese gustado haber tenido el tiempo suficiente para diseñar e implementar una inteligencia grupal con múltiples capas como la de *Killzone.* Con una primera capa de IA individual, otra a nivel de patrulla o pelotón y una última a nivel estratégico y táctico.

La primera capa debería ser muy similar a la que ya hemos implementado. Esta capa debería controlar a un único agente y ser capaz de recibir ordenes de las capas superiores.

La segunda capa debería actuar sobre un nivel más alto. Probablemente controlaría entre entre dos y cinco agentes que se encuentren en posiciones cercanas. Esta capa debería ser capaz gestionar y controlar a todos estos agentes para conseguir objetivos comunes mediante a creación de operaciones tácticas cooperativas de cobertura, ataque, supresión o flanqueo, sobre un personaje o área.

La ultima capa generaría ordenes a un nivel todavía más alto y seria la encargada de elaborar la estrategia global de la IA. Esta estrategia estaría formada por tres conceptos: mantener la posición, retroceder y avanzar. Esta capa gobernaría todo el sistema de IA y limitaría a los agentes a actuar de acuerdo a un objetivo global.

10.2 Supresión

Otro de los aspectos pendientes del proyecto es el fuego de supresión. Aunque su funcionamiento es correcto también es mejorable. La supresión se ha implementado como un comportamiento individual y el comportamiento grupal emerge a partir del desarrollo de estos comportamientos individuales.

La única manera para conseguir un comportamiento de supresión correcto es invertir tiempo en diseñar un sistema (como el explicado en el apartado anterior) capaz de coordinador a diversos agentes para que mientras uno avanza el otro permanezca suprimiendo o si un agente necesita recargar que el otro lo cubra, etc.

10.3 Mapa de influencia

Como hemos podido ver a lo largo del proyecto, la única fuente de influencia en el juego es el jugador. Creemos que se podría mejorar el sistema si los agentes inteligentes también fueran fuentes de influencia y de esta manera poder estar

informados de que zonas controlan.

Ademas, como ya cometamos previamente, el resultado de utilizar el mapa de influencia para realizar la predicción del jugador ha sido muy satisfactorio. Por esta razón, creemos que podríamos re-utilizar su implementación con pequeñas modificaciones para crear un mapa táctico o histórico de combate [17]. Estos mapas podrían ayudarnos a identificar que zonas controla cada equipo, las posiciones más desfavorables o los mejores flancos para atacar.

En general, consideramos que podríamos explotar aun más la implementación de los mapas de influencia con diferentes usos con el fin de mejorar significativamente el proceso de toma de decisiones y por lo tanto la experiencia del jugador.

10.4 Diversidad

Una de las ideas iniciales que teníamos al iniciar el proyecto era crear diferentes agentes con diferentes características tal y como pasaría en una situación real. Las características comprenderían desde cualidades físicas como la velocidad, vida, tamaño, tipo de arma (rifle, ametralladora, escopeta, etc) hasta cualidades más mentales como la aceptación de riesgos, posiciones o estados preferidas, etc.

El objetivo de incluir diferentes agentes con características únicas es explotar la diversidad y obtener el máximo beneficio de todas estas características. Ademas de mostrar un comportamiento menos uniforme y por lo tanto más real, este entorno facilitaría que emergiesen comportamientos cooperativos diferentes creando una experiencia de juego única.

10.5 Fuego amigo

Pese a haber tratado este problema durante el desarrollo del proyecto aun no esta completamente solucionado. Aunque resolvimos el conflicto que surgía cuando un agente se posicionaba en medio de una linea de tiro de otro agente no estamos tratando la fuente de este problema. Idealmente, los *NPCs* deberían evitar cruzar las lineas de fuego amigas.

Una solución válida y fácil de implementar seria incrementar enormemente el coste de atravesar las lineas de fuego amigas en el algoritmo de *pathfinding.* De esta manera, los agentes siempre buscarían un camino alternativo que no cruzara las lineas de fuego.

10.6 Malla de navegación dinámica

Una idea muy interesante que nos hubiese gustado incluir en el proyecto son los entornos dinámicos.

radicalmente como paredes que se pueden destruir, coches que se pueden mover o incluso edificios enteros que se pueden derribar. Todas estas acciones tienen una influencia estratégica y táctica en el mapa muy importante. Las posiciones de ataque y cobertura pueden cambiar o desaparecer así como pueden aparecer otras nuevas.

Por lo tanto, por un lado se necesita un sistema de IA que sea capaz de entender como estas acciones afectan al entorno y utilizarlas de manera adecuada a su favor en situaciones de combate. Por otro lado, el sistema también debe ser adaptativo y capaz de percibir de manera rápida todo tipo de cambios en el entorno, generados por el o por otro personaje, y modificar su comportamiento para obtener el máximo beneficio de la nueva situación.

Uno de los sistemas más afectados por estos cambios en el entorno es el sistema de navegación. La malla de navegación tiene que estar preparada para ser modificada, cambiando todos los costes y caminos tan pronto como un objeto (de tamaño considerable) se mueva o se destruya.

En nuestro caso utilizábamos una imagen 2D generada a partir de una vista superior del mapa para hacer algunas comprobaciones sobre la *Navmesh* (apartado 6.4.4.1) por lo que deberíamos actualizar esta imagen de manera frecuente (cada segundo) para que la IA sea capaz de detectar los cambios rápidamente.

Otro de los sistemas afectados significativamente por los cambios del entorno es el sistema de visibilidad. Dependiendo de la implementación del sistema, podemos encontrarnos con que debemos cambiar algunas estructuras o algoritmos. Aunque en nuestro caso no tendríamos que hacer ninguna modificación en el sistema de visibilidad ya que se calcula siempre en tiempo de ejecución, en otros juegos si hace falta. En *Killzone* por ejemplo, utilizan *cubemaps,* estructuras calculadas *offline* para acelerar los cálculos de visibilidad [19]. Estructuras que deberían ser modificadas si el entorno cambia.

10.7 Mirar alrededor

Un tarea bastante fácil de agregar al sistema de inteligencia artificial seria dotar a los *NPCs* de la capacidad de mirar alrededor cuando buscan o patrullan. Ahora mismo los *NPCs* siempre caminan mirando hacia delante o a su objetivo, si tienen uno. Esto no resulta real, sobretodo durante el comportamiento de búsqueda.

La idea entonces, seria que cuando el agente busca o patrulla, vaya mirando hacia los lados (izquierda y derecha) de manera intermitente para comprobar la mayor cantidad de posiciones posibles. Al fin y al cabo, así es como actuamos los seres humanos en situaciones de búsqueda. A medida que nos desplazamos, giramos la cabeza a lado y a lado para cubrir la mayor cantidad posible de terreno.

10.8 Predicción del jugador

Cuando los agentes pierden de vista al jugador utilizan como última posición (e inicio de la predicción en el mapa de influencia) allí donde lo han visto por última vez. Consideramos que este razonamiento no es adecuado y que realizar una predicción seria más correcto.

Los agentes deberían ser capaces de predecir la posición del jugador a partir de la última vez que lo vieron, su velocidad y dirección en aquel momento. De esta manera, cuando el jugador se esconde, los agentes tendrán una ligera idea de donde podría estar. Ademas, sería positivo que incorporasen esta información al mapa de influencia para que la influencia se comenzara a propagar desde esta nueva posición y que se fomentara la propagación de la influencia en aquella dirección y velocidad, incrementando la probabilidad de encontrar al jugador en esa zona. Así, el sistema en general, y especialmente el de búsqueda, experimentaría una mejora notable.

10.9 Rendimiento

Por último, aunque no experimenté ningún problema de rendimiento, creo que algunos de los algoritmos que utilice podrían mejorarse,¡ así como crear estructuras de datos para mejorar el rendimiento.

Creemos que se podría mejorar el algoritmo de visibilidad. Aunque conseguimos reducir mucho el número de rayos necesarios para calcular la visibilidad en comparación con otras técnicas, consideramos que en algunas situaciones se podrían reducir ligeramente el número de rayos trazados. Por ejemplo, cuando un objeto esta detrás de otro, no hace falta trazar los rayos al objeto de detrás porque ya sabemos que no se vera.

Por otro lado, como hemos visto a lo largo del proyecto el algoritmo de visibilidad (entre otros) se utiliza en diversas ocasiones. Por esta razón, creemos que seria positivo utilizar una cache o una estructura de datos auxiliar para evitar ejecutar el algoritmo otra vez si lo ejecutamos recientemente.

11

Referencias

Bibliografía

1: Wikipedia, Artificial Intelligence, 2016, https://en.wikipedia.org/wiki/Artificial_intelligence
2: AiGameDev.com, Nucl.ai courses: The Principles of Modern Game AI, 2016, https://courses.nucl.ai/
3: Kevin Dill, Damian Isla, Neil Kirby, Dave Mark, Steve Rabin, Nathan Sturtevant and Simon Tomlinson , Game AI Pro: Collected Wisdom of Game AI Professionals, 2013, http://www.gameaipro.com/
4: Alex J. Champandard, On Finite State Machines and Reusability, 2007, http://aigamedev.com/open/article/fsm-reusable/
5: Alex J. Champandard, The Gist of Hierarchical FSM, 2007, http://aigamedev.com/open/article/hfsm-gist/
6: Alex J. Champandard, Understanding Behavior Trees, 2007, http://aigamedev.com/open/article/bt-overview/
7: Damián Isla, Building a Better Battle: The Halo 3 AI Objectives System, 2008, http://web.cs.wpi.edu/~rich/courses/imgd4000-d09/lectures/halo3.pdf
8: Alex J. Champandard, The Power of Sequences for Hierarchical Behaviors, 2007, http://aigamedev.com/open/article/sequence/
9: Alex J. Champandard, The Flexibility of Selectors for Hierarchical Logic, 2007, http://aigamedev.com/open/article/selector/
10: Alex J. Champandard, Using Decorators to Improve Behaviors, 2007, http://aigamedev.com/open/article/decorator/
11: Jeff Orkin, Three States and a Plan: The A.I. of F.E.A.R, 2006, http://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf
12: Gerhard Wickler and Austin Tate, Artificial Intelligence Planning, , https://class.coursera.org/aiplan-003/lecture
13: William van der Sterren, Multi-Unit Planning with HTN and A*, 2009, http://www.cgf-ai.com/docs/multi_unit_htn_with_astar.pdf
14: Noah Brickman and Nishant Joshi, HTN Planning and Game State Management in Warcraft II, , http://www.cs.uu.nl/docs/vakken/b2ki/Docs/Literatuur/SHOPforWarcraft.pdf
15: Arno Kamphuis, Michiel Rook, Mark H. Overmars, Tactical Path Finding in Urban Environments, , http://www.cs.uu.nl/groups/AA/movie/publications/PDF/KRO05.pdf
16: Remco Straatman, William van der Sterren and Arjen Beij, Killzone's AI: Dynamic Procedural Tactics, 2005, http://www.cgf-ai.com/docs/killzone_ai_gdc2005_slides.pdf
17: Alex J. Champandard, The Mechanics of Influence Mapping: Representation, Algorithm & Parameters, 2011, http://aigamedev.com/open/tutorial/influence-map-mechanics/
18: Remco Straatman, Tim Verweij, Alex Champandard,Robert Morcus, and Hylke Kleve, Hierarchical AI for MultiplayerBots in Killzone 3, 2013, http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter29_Hierarchical_AI_for_Multiplayer_Bots_in_Killzone_3.pdf
19: Alex J. Champandard and Michiel van der Leeuw, Waypoint Cover Maps and Efficient Raycasts on PS3 in Killzone 2, 2009, http://aigamedev.com/insider/coverage/waypoint-cover-maps/

20: Alex J. Champandard , Open Challenges in First-Person Shooter (FPS) AI Technology, 2011, http://aigamedev.com/open/editorial/open-challenges-fps/
21: Alex Champandard, Tim Verweij and Remco Straatman, Killzone 2 Multiplayer Bots, , http://es.slideshare.net/guerrillagames/killzone-2-multiplayer-bots
22: Valve Developer Community, Source (Game Engine), 2015, https://developer.valvesoftware.com/wiki/Main_Page
23: Crytek, Cry Engine (Game Engine), 2016, https://www.cryengine.com/
24: Wikipedia, AAA (video game industry) - Wikipedia, 2016, https://en.wikipedia.org/wiki/AAA_(video_game_industry)
25: Eddie Makuch, Amazon and Crytek Agree to licensing deal worth $50-$70 Million, 2015
26: Unity Technologies, Unity 3D (Game Engine), 2016, https://unity3d.com/es
27: Anderson Cardoso, Behaviour Machine Free, 2016, https://www.assetstore.unity3d.com/en/#!/content/20280
28: Unity, Survival Shooter Tutorial, 2016, https://unity3d.com/es/learn/tutorials/projects/survival-shooter-tutorial
29: Epic Games, Unreal Engine 4 (Game Engine), 2016, https://www.unrealengine.com/what-is-unreal-engine-4
30: Unreal Engine 4, AI Perception (UE4 Official Docs), 2016, https://docs.unrealengine.com/latest/INT/BlueprintAPI/AI/Perception/index.html
31: Unreal Engine 4, Navmesh Content Examples (UE4 Official Docs), 2016, https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/NavMesh/index.html
32: Unreal Engine 4, Navigation Components (UE4 Official Docs), 2016, https://docs.unrealengine.com/latest/INT/Engine/Components/Navigation/
33: Unreal Engine 4, Behavior Trees (UE4 Official Docs), 2016, https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/index.html
34: Unreal Engine 4, Environment Query System (UE4 Official Docs), 2016, https://docs.unrealengine.com/latest/INT/Engine/AI/EnvironmentQuerySystem/index.html
35: Epic Games, Unreal Engine Source Code, 2016, https://github.com/EpicGames/UnrealEngine
36: Red Blob Games, 2D Visibility, 2012, http://www.redblobgames.com/articles/visibility/
37: Blackpawn, Point In Triangle Test, , http://www.blackpawn.com/texts/pointinpoly/
38: Mieszko, AI Perception only can detect middle of player, 2015, https://answers.unrealengine.com/questions/229043/ai-perception-only-can-detect-middle-of-player.html
39: Envato Tuts+, Understanding Steering Behaviors, 2012, http://gamedevelopment.tutsplus.com/series/understanding-steering-behaviors--gamedev-12732
40: Mieszko Zielinski, Unreal Engine AI Tutorial - 2 - Avoidance, 2014, https://wiki.unrealengine.com/Unreal_Engine_AI_Tutorial_-_2_-_Avoidance
41: Jur van den Berg, Ming Lin and Dinesh Manocha, Reciprocal Velocity Obstacles for Real-Time Multi-Agent Navigation, 2008, http://gamma.cs.unc.edu/RVO/icra2008.pdf

42: Mtrebi, MilkyEngineer, Modify pathfinding cost - Unreal Answers, 2015, https://answers.unrealengine.com/questions/351864/modify-pathfinding-cost-detournavmeshquery-subclas.html

Índice de vídeos

Vídeo 1: Demostración del algoritmo de propagación de influencia. Enlace http://youtu.be/VE70oB4xyrw.....126
Vídeo 2: Demostración del algoritmo de propagación de influencia teniendo en cuenta la visibilidad de un NPC. Enlace http://youtu.be/j0cO867Jg7A.....127
Vídeo 3: Demostración del comportamiento de patrulla. Enlace http://youtu.be/GxMbbNWw-DA.....128
Vídeo 4: Demostración del comportamiento de búsqueda. Enlace http://youtu.be/fTocYPT-k6o.....128
Vídeo 5: Demostración del comportamiento de cobertura final. Enlace http://youtu.be/EW3r7214_X0.....129
Vídeo 6: Demostración del comportamiento de cobertura simplificado. Enlace http://youtu.be/gyrZ40zJT3I.....130
Vídeo 7: Demostración del comportamiento de avance. Enlace http://youtu.be/ibkQxci0enk.....130
Vídeo 8: Demostración del comportamiento de supresión. Enlace http://youtu.be/JWuTSY9xDPg.....131
Vídeo 9: Demostración del comportamiento de ataque. Enlace http://youtu.be/wSTioum0eas.....132
Vídeo 10: Demostración de un ataque en grupo. Enlace http://youtu.be/zs1O1GdHL7U.....132
Vídeo 11: Demostración de un jugador luchando contra un agente. Enlace http://youtu.be/sQ0by2-ijB8.....134
Vídeo 12: Demostración de un jugador luchando contra cuatro agentes. Enlace http://youtu.be/TgmPYbJQutw.....135

Índice de tablas

Tabla 1: La temporalización inicial de las diferentes fases del proyecto.....11
Tabla 2: Evaluación de los requisitos necesarios por parte del motor Unity 3D.....40
Tabla 3: Evaluación de los requisitos necesarios por parte del motor Unreal Engine 4 41
Tabla 4: Comparativa de las características necesarias para el proyecto de Unity 3D y Unreal Engine 4.....42
Tabla 5: Memoria del árbol de comportamiento principal.....53
Tabla 6: Memoria del árbol de comportamiento Patrol.....56
Tabla 7: Memoria del árbol de comportamiento Search.....58
Tabla 8: Memoria del árbol de comportamiento Fight.....60

Índice de figuras

Figura 2.1: Esquema describiendo el proceso de Pensar-Sentir-Actuar de un agente inteligente....14
Figura 2.2: Esquema FSM representando el comportamiento de un guardia....16
Figura 2.3: Esquema FSM representando un comportamiento simple de patrulla....17
Figura 2.4: Esquema FSM con múltiples instancias del estado Conversation....17
Figura 2.5: Esquema de una HFSM solucionando la duplicidad de estados....18
Figura 2.6: BT formado por diversos comportamientos modulares....19
Figura 2.7: Nodo compuesto de tipo secuencia formada por cuatro comportamientos 20
Figura 2.8: Nodo compuesto de tipo selector con tres hijos que está ejecutando el segundo hijo....21
Figura 2.9: Nodo compuesto de tipo paralelo ejecutando dos comportamientos simultáneamente....21
Figura 2.10: Pseudocódigo simplificado de un planificador GOAP....24
Figura 2.11: Pseudocódigo simplificado de un planificador que utiliza HTN....26
Figura 2.12: Propagación de la influencia a partir de un nodo con 1 de influencia....28
Figura 2.13: Posiciones visibles por un agente....29
Figura 2.14: Killzone 2 utiliza waypoints con cubemaps (arriba a la derecha) pre-cálculados para determinar la visibilidad de cada zona....30
Figura 2.15: Esquema de un sistema de IA con múltiples capas....32
Figura 2.16: Sistema de inteligencia por capas de Killzone 2....33
Figura 3.1: Unity tutorial Survival Shooter....39
Figura 3.2: UE4 tutorial FPS....41
Figura 4.1: Sistema de percepción de sonido con un radio determinado en un NPC 44
Figura 4.2: Navmesh generada por UE4 (en color verde)....45
Figura 4.3: Navmesh generada por UE4 (en color verde) incluyendo un enlace navegable....46
Figura 4.4: Navmesh generada por UE4 (en color verde) incluyendo un incrementador de coste de manera manual (en color rojo)....47
Figura 4.5: Herramienta visual para diseñar Behavior Trees en UE4....48
Figura 4.6: Elementos generados por una consulta EQS con forma de rejilla y un único test de distancia que favorece los elementos cercanos. El número al costado de cada elemento es su puntuación (más grande mejor)....49
Figura 5.1: Árbol de comportamiento principal....54
Figura 5.2: Árbol de comportamiento Idle....55
Figura 5.3: Árbol de comportamiento Patrol....57
Figura 5.4: Árbol de comportamiento Search....58
Figura 5.5: Árbol de comportamiento simplificado de Fight....61
Figura 5.6: Condición de activación del comportamiento Hide....62
Figura 5.7: Rama de comportamiento correspondiente a la cobertura....63
Figura 5.8: Condición de activación del comportamiento Push....64
Figura 5.9: Rama de comportamiento correspondiente al avance....65
Figura 5.10: Condición de activación del comportamiento Attack....66
Figura 5.11: Árbol de comportamiento correspondiente a la rama Attack....68

Figura 5.12: Diagrama de estados describiendo el comportamiento de la variable State....71
Figura 5.13: Calculo de la visibilidad del agente respecto del jugador....72
Figura 5.14: Algoritmo A* con una heurística basada en distancia euclidiana. Polígonos evaluados y camino elegido....76
Figura 5.15: Puntos de patrulla de un NPC. A la izquierda la lista de puntos y a la derecha los puntos situados en el espacio....77
Figura 5.16: Resultado de la consulta EQS de búsqueda....78
Figura 5.17: Resultado de la consulta EQS de cobertura....79
Figura 5.18: Resultado de la consulta EQS de ataque....81
Figura 5.19: Resultado de la consulta EQS de avance....82
Figura 5.20: Resultado de una consulta EQS para obtener los puntos de supresión 83
Figura 5.21: Resultado de una consulta EQS para obtener nuevos puntos con linea de tiro....84
Figura 6.1: Algoritmo para determinar la visibilidad haciendo Raytrace cada un cierto ángulo....87
Figura 6.2: Algoritmo para determinar la visibilidad haciendo Raytrace unicamente a los vértices....87
Figura 6.3: Inicio y fin de la zona de visibilidad....88
Figura 6.4: Zona de visibilidad por un personaje determinada por el rango y angulo de visión....88
Figura 6.5: Puntos delimitadores de visibilidad ordenados....88
Figura 6.6: Situación 1. Al trazar el primer rayo encontramos una colisión con un obstáculo....89
Figura 6.7: Situación 2. Al trazar el primer rayo no hay colisión....89
Figura 6.8: Situación 3. El rayo impacta antes de alcanzar el vértice. Triangulo de visibilidad determinado en verde....90
Figura 6.9: Situación 4 Al trazar un rayo no impactamos en el vértice. Triangulo de visibilidad determinado en verde....91
Figura 6.10: Situación 5 Al trazar el un rayo no impactamos en el vértice. Triangulo de visibilidad determinado en verde....91
Figura 6.11: Trazado de rayos y construcción de los triángulos de visibilidad....92
Figura 6.12: Visibilidad final determinada por la unión de los triángulos....92
Figura 6.13: Código del algoritmo para calcular la visibilidad....92
Figura 6.14: Código de la función auxiliar que determina los triángulos de visibilidad 93
Figura 6.15: Visibilidad determinada (color negro) trazando unicamente rayos a los vértices de los obstáculos....94
Figura 6.16: Camino más corto determinado por un algoritmo A* con una heurística basada unicamente en distancia euclidiana....95
Figura 6.17: Camino más seguro determinado por un algoritmo A* con una heurística basada en distancia euclidiana y visibilidad del jugador....95
Figura 6.18: Algoritmo 2 para realizar el cálculo del Tactical Pathfinding. Segmento divido en varios puntos. En rojo los puntos más costosos (visibles) y en verde los menos....96
Figura 6.19: División de un segmento en N nuevos puntos equidistantes....97

Figura 6.20: Código del algoritmo de división de un segmento.....97
Figura 6.21: Código del algoritmo 2 completo. Cálculo de coste utilizando un modificador.....98
Figura 6.22: Ccódigo del algoritmo 2 completo final. Cálculo del coste basado en distancia visible contínua.....99
Figura 6.23: Mapa de influencia representado utilizando una matriz 2D.....100
Figura 6.24: Código del algoritmo de propagación de influencia.....102
Figura 6.25: Codificación del mapa en una imagen 2D. Las zonas navegables están marcadas en verdes y las no navegables en negro.....103
Figura 6.26: Vecinos de una posición en una matriz.....104
Figura 6.27: Código de la función de disparar y suprimir.....105
Figura 6.28: Situación de combate con un agente AI 2 (señalado en rojo) sufriendo fuego amigo.....105
Figura 6.29: Código de la función de cálculo linea de fuego.....106
Figura 6.30: Resultado de una consulta EQS con un único test de coste favoreciendo las posiciones cercanas.....107
Figura 6.31: Resultado de una consulta EQS con un único test de distancia de un personaje a todos los puntos del mapa.....108
Figura 6.32: Resultado de una consulta EQS con un único test de visibilidad.....109
Figura 6.33: Demostración de la formula de puntuación utilizada en el test.....109
Figura 6.34: Resultado de una consulta EQS con un único test de detrás de los obstáculos.....110
Figura 6.35: Resultado de una consulta EQS con un único test de proximidad a los obstáculos.....111
Figura 6.36: Resultado de una consulta EQS con un único test de influencia.....112
Figura 6.37: Resultado de una consulta EQS con un único test de expansión favoreciendo las posiciones alegadas del agente AI 1.....113
Figura 6.38: Demostración de la formula utilizada en el test de flanqueo.....114
Figura 6.39: Resultado de una consulta EQS con único test de flanqueo.....114
Figura 7.1: Situación problemática en la que la IA no detecta que el jugador es visible 117
Figura 7.2: Solución utilizada para detectar la visibilidad a través del uso de tres rayos (cabeza y hombros).....118
Figura 7.3: Situación problemática en la cual los NPCs se han quedado atascados 119
Figura 7.4: VO de un obstáculo B respecto un agente A.....120
Figura 7.5: RVO de un obstáculo B respecto un agente A.....121
Figura 7.6: A la izquierda el mapa original y a la derecha la textura de baja resolución (64x64) utilizada para la depuración.....122
Figura 7.7: Propagación de la influencia alrededor del jugador.....123