Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Máster Universitario en Inteligencia Artificial

Trabajo Fin de Máster

Inteligencia Artificial Basada en Multiagentes para Aplicaciones Interactivas Multimedia

Autor(a): Jorge Pablo Yanguas Martín
Tutor(a): Javier Bajo Pérez

Madrid, Febrero 2021

Este Trabajo Fin de Máster se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Máster
Máster Universitario en Inteligencia Artificial

Título: Inteligencia Artificial Basada en Multiagentes para Aplicaciones Interactivas Multimedia

Febrero 2021

Autor(a): Jorge Pablo Yanguas Martín
Tutor(a): Javier Bajo Pérez
Departamento de Inteligencia Artificial
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

En este trabajo se analizarán las ventajas e inconvenientes de las herramientas utilizadas actualmente para la creación de inteligencia artificial utilizadas para videojuegos por parte de empresas de gran influencia en este sector. Estas herramientas son Unreal Engine 4, Unity 3D y Source Engine. Se analizarán particularmente estas herramientas debido a que tanto las herramientas como su documentación están disponibles para el libre acceso gratuito de cualquier persona, y actualmente están ampliamente utilizadas y bastante extendidas desde el punto de vista comercial.

Utilizando y comparando la información recopilada mencionada en el párrafo anterior, se expondrá la necesidad de aplicar nuevas técnicas en estos ámbitos, y se procederá y a explicar en profundidad los requisitos y razones que han guiado el diseño, el funcionamiento, el comportamiento y los beneficios de la inteligencia artificial basada en multiagentes creada para el videojuego Executanks.

Abstract

This works proceeds to analyze benefits and drawbacks from actual tools that are currently used to create videogame artificial intelligence by influential companies that are already established on this sector. This tools are Unreal Engine 4, Unity 3D and Valve Hammer Editor. This tools will be particulary analyzed due both tools and their respectively documentations are available for free to any user and because their use is currently widely extended commercially speaking.

Using and comparing the information collected and mentioned in the previous paragraph, this work will expose the needing of new techniques on this scope, and deeply explain the requirements and the reasons that ruled the design, the behavior and the benefits of multi agent based artificial intelligence created for the Executanks video game.

Tabla de contenidos

1. Introducción **1**

2. Desarrollo **3**
2.1. Introducción . . . 3
2.2. Estado del arte . . . 3
2.2.1. GOAP Goal Oriented Action Planning . . . 3
2.2.2. Behavior Trees o Árboles de Comportamiento . . . 5
2.2.2.1. Behavior Trees en Unreal Engine4 . . . 8
2.2.3. Sistemas multi-agente y sistemas multi-agente auto-adaptativos (AMAS) . . . 9
2.2.3.1. Sistemas multi-agente . . . 9
2.2.3.2. Sistemas multi-agente auto adaptativos . . . 9
2.2.3.3. Sistemas multi-agente en aplicaciones interactivas . . . 10
2.2.4. Matrices de interacción basadas en reglas . . . 11
2.2.4.1. IODA . . . 12
2.2.4.2. The Galaxian Project . . . 13
2.2.4.3. FormatStore . . . 13
2.2.5. Algoritmo HPA* . . . 15
2.2.6. Malla de Navegación o NavMesh . . . 17
2.2.6.1. NavMesh en Unity . . . 18
2.2.6.2. Áreas del NavMesh en Unity . . . 19
2.2.6.3. Agentes del NavMesh en Unity . . . 20
2.2.6.4. NavMesh en Unreal Engine 4 . . . 22
2.2.6.5. NavMesh en Source Engine . . . 23

3. Propuesta **25**
3.1. Desarrollo de la inteligencia artificial de Executanks . . . 25
3.1.1. Requisitos . . . 25
3.1.2. Entorno . . . 26
3.1.3. Descripción de los Agentes utilizados . . . 27
3.1.4. Descripción del comportamiento . . . 30
3.1.5. Control de velocidad . . . 31
3.1.6. Control de giro . . . 32
3.1.7. Control de la torreta . . . 33
3.1.8. Comportamientos inesperados manifestados . . . 33

4. Resultados y conclusiones **37**

Bibliografía **41**

Anexo **42**

Capítulo 1

Introducción

Una de las principales tareas del desarrollo de un videojuego o entorno virtual interactivo y que además es de las más difíciles, es el desarrollo de una inteligencia artificial de forma que cumpla los propósitos de la aplicación y amenudo con restricciones de hardware.

En este documento se analizarán algunos de los métodos más conocidos utilizados por los motores de videojuegos como pueden serlo Unreal Engine 4, Unity 3D o Source Engine 2.

Se comentarán los algoritmos y herramientas utilizados por estas herramientas que por normal general requieren para su funcionamiento estructuras de datos precalculadas como lo son el algoritmo HPS*, los PathNodes y NavMesh o mallas de navegación, debido tanto al interés académico como al comercial que tienen estos métodos actualmente. Y por otra parte se analizarán también los aspectos y beneficios que aportan otro tipo de aproximaciones como lo son los métodos basados en multiagentes.

En el último apartado tras haber analizado los pros y contras de los métodos actuales, y haber escogido aquellos aspectos que se adaptan mejor a las necesidades de los requisitos planteados por el videojuego Executanks, se procederá a explicar en detalle las razones sobre el desarrollo de una inteligencia artificial adaptativa basada en multiagentes, que no requiere estructuras de datos precalculadas, y que se adapta a un entorno cambiante. Tampoco requiere un mapa de rejilla (tile based) para funcionar, Y que además soporta en tiempo real una gran cantidad de obstáculos y agentes controlados por inteligencia artificial.

Capítulo 2

Desarrollo

2.1. Introducción

En este documento se analizan diversos artículos sobre inteligencia artificial que son interesantes desde el punto de vista de desarrollo de videojuegos, ya sea por que han sido utilizados anteriormente en ámbitos similares, o por que ofrecen puntos de vista o funcionalidades interesantes en este ámbito, ya sea por la posibilidad de ser implementados y ejecutados en tiempo real como el algoritmo HPA*, o por su capacidad de definir comportamientos lo más parecido posible a jugadores reales o a otro tipo de agentes interactivos.

Además se analizan herramientas de desarrollo de videojuegos. Más concretamente sus funcionalidades sobre pathfinding y las posibilidades que nos ofrecen sus agentes para el desarrollo de una inteligencia artificial dedicada a los propósitos que pueda necesitar el usuario. Además, se analizan los pros y contras de estas herramientas que a fin de cuentas fueron las razones que motivaron el desarrollo de la inteligencia artificial basada en multiagentes de Executanks.

2.2. Estado del arte

A lo largo del tiempo en el que la industria de los videojuegos ha estado en marcha se han utilizado diferentes técnicas de artificial aplicadas a esta industria. Estas técnicas han ido respondiendo a las necesidades de los proyectos en función de su diseño. Posiblemente algunas de las técnicas más conocidas en este ámbito son las máquinas de estados, la planificación de acciones orientada a objetivos, los sistemas multiagente y los árboles de comportamiento, siendo estos últimos al parecer bastante populares[15] además de que herramientas como Unreal Engine 4 y Unity3D disponen herramientas integradas o plugins para desarrollar este tipo de inteligencia artificial.

2.2.1. GOAP Goal Oriented Action Planning

Posiblemente entre las técnicas más usadas para inteligencia artificial de videojuegos se encuentran el algoritmo A* y las máquinas de estados finitos [17]. Jeff Orkins en su artículo explica como él mismo, para su implementación GOAP, utiliza una máquina de estados finitos de tan solo tres estados. El estado **Animation** que

ejecutaría una animación,otro estado **GoToPosition** que indicaría al agente moverse hacia una posición, y por último **UseSmartObject** que sería el encargado de utilizar un objeto "smartçomo una puerta. Y el algoritmo A* para desarrollar el planificador. Los beneficios principales que podemos obtener al usar Goal Oriented Action Planning son la obtención de comportamientos más variados, complejos e interesantes. Además el código que implementa estos comportamientos, es más estructurado, es más mantenible, escalable y reutilizable[16].

Podemos definir la planificación de acciones orientada a objetivos como una arquitectura de toma de decisiones que además de determinar qué hacer, determina la forma de cómo hacerlo.[16]

Este tipo de inteligencia artificial se basa en los conceptos **Goal**(objetivo), **Action**(acción), **Plan**(el plan) y **Formulate**(formular)

Conceptos:

- **Goal**: Es el objetivo de la inteligencia artificial. Es aquello que representa lo que el agente desea que ocurra. Es posible tener diferentes objetivos con diferentes prioridades para que se efectuen unos antes que otros.
- **Action**: Las acciones son los posibles pasos que se requieren para cumplir un objetivo. Estos pasos deben cumplirse en su totalidad(atómicos), para pasar al siguiente paso. La duración de los pasos pueden ser variable desde ser muy corta, como puede ser el hecho de abrir una puerta, o posiblemente infinita como el hecho de perseguir o atacar a un jugador.[16]
- **Plan**: Es la secuencia de acciones que permite el desarrollo del plan.
- **Formulate**: Es el hecho de formular un plan. Una posible forma de hacer esto es que un agente controlado por inteligencia artificial le de información sobre su estado y el estado del objetivo para generar un plan.

Una problemática a la que se tienen que enfrentar este tipo de sistemas es la posibilidad de que un plan deje de ser válido mientras está siendo ejecutado. Esto puede ocurrir debido a que pueden coexistir en el mismo entorno interactivo varios agentes incluído el jugador, que pueden alterar los elementos del escenario. De este modo, los agentes de inteligencia artificial deben verificar si su plan sigue siendo ejecutable durante la ejecución del mismo. En caso de que un agente quede con un plan invalidado éste debe pedir al planificador un nuevo plan[16].

2.2.2. Behavior Trees o Árboles de Comportamiento

Los Behavior Trees o Árboles de Comportamiento son una forma de organizar las diferentes acciones o tareas que puede ejecutar un sistema controlado por inteligencia artificial. Ya sea un robot o como es el caso de este trabajo, un agente virtual en un videojuego. [14]

Como se puede ver en la figura 2.1, en este caso las secuencias de acciones se representan en forma de árbol con distintos nodos que representarían las acciones de la secuencia. La secuencia representada por el árbol muestra las acciones necesarias para recoger una pelota. Que requiere las acciones. Encontrarla, cogerla, y finalmente colocarla. Como podemos ver, cada nodo representa una acción. Y cada nodo puede dividir sus acciones en acciones más detalladas representado de forma más precisa el comportamiento que se desea. En la figura mencionada se puede observar como la acción "Pick Ball"se divide en acciones más pequeñas.

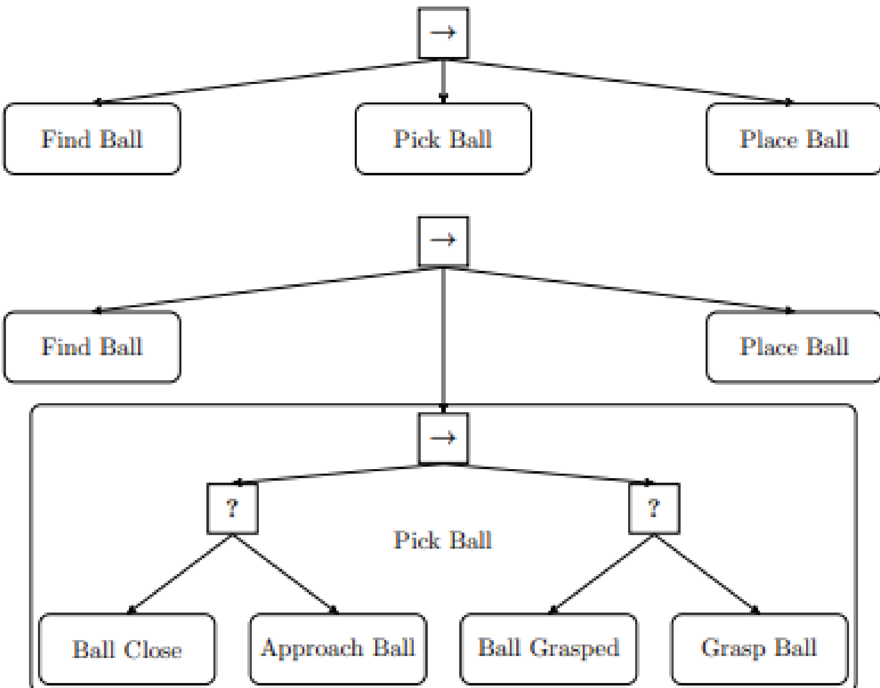

Figura 2.1: Ejemplo de un árbol de comportamiento o behavior tree. Imagen tomada de la referencia [14]

Los behavior trees surgen en la industria de los videojuegos como una herramienta para mejorar la modularidad en las estructuras de control de los agentes controlados por inteligencia artificial de éstos mismos [14]. Los behavior trees nos ofrecen la posibilidad de modularizar el diseño de los comportamientos de los agentes controlados por inteligencia artificial con el nivel de detalle que deseemos. Este tipo de estructura nos permite mediante las estructuras de control de los árboles definir un comportamiento tan específico como deseemos.

Los behaviorTrees son similares a las máquinas de estados finitos desde el punto de vista en el que ambos van pasando de un nodo a otro que representan las diferentes tareas o acciones. Con un behaviorTree bien diseñado es posible representar mediante sus jerarquías y ramificaciones, comportamientos bastante complejos

creados a partir de tareas sencillas. Al contrario que las máquinas de estados finitos, los behaviorTrees están formados por "Behaviors" o comportamientos más que de los estados en sí. Es decir: el árbol representa directamente el comportamiento. Además, el hecho de que los behaviorTrees sean fáciles de diseñar, probar y debugear los han convertido en una herramienta bastante popular desde su uso en conocidos videojuegos como Halo 2 y Bioshock [12]. De hecho podemos ver como Epic Games en su motor Unreal Engine 4 integra de manera nativa la posibilidad de crear nuestos propios behabiorTrees en nuestras apicaciones[38].

Habitualmente durante la ejecución de un motor de videojuego implica el procesamiento de pequeños pasos para diferentes propósitos como pueden serlo la ejecución del motor de físicas, la comprobación de las normas del juego y por supuesto los cómputos necesarios para el procesamiento de la inteligencia artificial. Los árboles de comportamiento se ejecutan también entre estos pasos que van comprobando los eventos ocurridos en el juego para ir cambiando entre las ramas en función de los eventos ocurridos en el videojuego. Los behaviorTrees se ejecutan empezando desde el nodo raíz ejecutando su acción asignada para luego ir avanzando por los hijos del nodo de izquierda a derecha. Éstos nodos, a su vez pueden tener más ramificaciones y más nodos hijos que se seguirán procesando siguiendo ese mismo orden.

Los behaviorTrees pueden tardar bastante tiempo en ejecutarse. Por esta razón se le da un tiempo de cpu para procesar sus acciones. Cuando este tiempo de cpu se agota se retorna un código de estado que da información sobre el estado de la ejecución del behaviorTree. Para poder controlar y gestionar si la inteligencia artificial necesita más tiempo o pasos para ejecutarse, se utilizan varios códigos de estado.

Posibles códigos de estado:

- **SUCCESS**: Se retorna este estado cuando la tarea representada por los componentes del behavior tree ha sido completada con éxito.
- **FAILURE**: Este estado es el que se retorna cuando la tarea ha sido completada pero no de forma satisfactoria. No confundir con ERROR. El hecho de que una acción retorne el estado FAILURE indica que el agente fué incapaz de terminar la acción, y no indica que haya habido un error en el código.
- **RUNNING**: En caso de que la inteligencia artificial no haya terminado y requiera más pasos para ejecutar su tarea devolverá este estado.
- **ERROR**: Este estado al contrario que FAILURE si se utiliza para encontrar errores en el código. Es decir, tiene propósitos de debugging. No debería ocurrir durante una ejecución normal.

Según el artículo mencionado, aunque dependiendo de la implementación de los behaviorTrees se pueden definir más componentes para su diseño, debería ser posible poder definir cualquier comportamiento complejo mediante el uso de los cinco componentes fundamentales enumerados a continuación. [21]

Componentes fundamentales:

- **Reference**: Este componente enlaza con otro behavior tree. Gracias a este componente, un nodo del árbol puede hacer referencia a otro y el código de estado del behavior tree referenciado pasará a este como si formaran parte del mismo. Este componente se representa mediante un cuadrado con doble línea con el

nombre del behavior tree al que enlaza.

- **Action**: El componente acción ejecuta algún tipo de función del juego. Ya sea ejecutar un sonido, cambiar el estado del agente, ejecutar alguna función de pathfinding para hacer algún recorrido en específico o algún tipo de lógica específica del juego. Una acción puede tardar varios fotogramas en ejecurarse. De este modo, es posible que se retornen los estados SUCCESS, FAILURE o RUNNING en función si la tarea ha sido terminada con éxito, sin éxito, o si todavía se está ejecutando. Este componente se representa mediante una caja de una línea con el nombre de la acción.

- **Condition**: Este componente almacena una condición que debe ser evaluada. Una condición retorna SUCCESS en caso de cumplirse, FAILURE en caso de no cumplirse. Ya que el componente condition no represnta una acción si no solo una pregunta, no retorna nunca el estado RUNNING. El componente condition se representa como una caja de la misma forma que la de acción pero con una interrogación al final.

- **Flow control**: El componente de control de flujo agrupa un conjunto de componentes hijos e indica su orden de ejecución. El comportamiento de este componente varía en función de los hijos que tenga. Existen dos tipos de este componente que son "selector" y "sequence".

- **Selector**: Recorre los hijos de izquierda a derecha buscando descartando aquellos que retornan de forma inmediata el estado FAILURE. Cuando encuentra un hijo que retorna el estado SUCCESS o en caso de que necesite más tiempo el estado RUNNING, pasa a procesar estos. Solo en caso de que todos los hijos retornen FAILURE el selector retorna dicho estado. Este nodo de control de flujo se representa como un círculo con una interrogación en medio.

- **Sequence**: La secuencia ejecuta sus componentes hijos de izquierda a derecha. Difiere del caso anterior en que inmediatamente si uno de los compnentes hijos devuelve el estado FAILURE. En caso de que los componentes retornen RUNNING o SUCCESS se siguen ejecutando de uno en uno. Hasta que todos los componentes hijos no han devuelto el estado SUCCESS indicando que la secuencia se ha completado, este componente no retorna dicho estado. Este control de flujo se representa como una caja con una flecha que va de izquierda a derecha.

- **Decorator**: El componente decorator modifica el comportamiento de uno de los componentes anteriores. Por ejemplo negando una condición, repitiendo un componente una cantidad determinada de veces o limitando el tiempo de una acción en particular. Es posible mediante decorators forzar el retorno de un estado en particular. También existen decorators que pueden llegar a tener pseudocódigo. Un componente puede tener tantos decorators como sean necesarios. Los componentes decorator se representan como un icono adicional en el interior de las cajas de los demás componentes.

Los behavior trees convencionales tienen ciertas limitaciones que algunos autores como veremos a continuación proponen ideas para solucionarlos. Una de estas pegas es la limitación que tienen los behavior trees para intercomunicarse crear comportamientos en grupo.

Los behavior trees tienen múltiples usos en diferes ámbitos. En el propio artículo citado [14] podemos ver ejemplos de cómo se puede implementar este tipo de estructuras para la creación de una inteligencia artificial para el conocido videojuego PACMAN, también podemos ver otros ejemplos en otras industrias como pueden ser robótica industrial o vehículos de transporte autónomos.

Actualmente los behavior trees han evolucionado para satisfacer nuevas necesidades como puede ser la cooperación entre diferentes agentes[15]. En el artículo mencionado se hace una proposición sobre como extender la capacidad de los Event Driven Behavior Trees e implementar una arquitectura controlada por eventos para este tipo de estructuras, que además da la capacidad de comuniación e interacción entre agentes mediante la creación de nuevos nodos. Estos nodos añaden la funcionalidad que permite este tipo de necesidades mediante el uso de los tres nuevos nodos llamados Coordination nodes o "nodos de coordinación" que pueden enviar y recibir mensajes a través de una bandeja de entrada o como la llaman en el artículo "mailbox" que es una cola de prioridad que almacena mensajes hasta que son seleccionados o descartados.

Nodos nuevos en los EDBT:

- **Request Handler Node**: Es el nodo encargado de seleccionar y gestionar un nuevo mensaje recibido a su bandeja de entrada.
- **Soft Request Sender Node**: Este nodo permite enviar un mensaje que contiene una condición que los receptores deben intentar cumplir además de un lapso de tiempo que puede expirar.
- **Hard Request Sender Node**: Este nodo tiene como hijo una raíz de un árbol que representa un comportamiento dependiente de algún cometido o tarea que requiere algún tipo de cometido por parte de los receptores.

A pesar de la funcionalidad añadida por la utilización de estas herramientas, en el propio artículo se comentan algunos problemas que tiene este sistema pero que piensan arreglar en el futuro. Como puede serlo añadir la capacidad para enviar peticiones a diferentes tipos de receptores de forma simultánea. Además, aunque se comenta el hecho de que se puede hacer mediante el uso de varias peticiones, también se plantean la posibilidad de diseñar un nodo que sea capaz de realizar peticiones en cadena.

2.2.2.1. Behavior Trees en Unreal Engine4

Unreal Engine 4 tiene integrado en la herramienta un editor de Behavior Trees. Aunque en la práctica siguen siendo árboles de comportamiento, en la propia documentación de Epic-Games nos comentan que existen diferencias en cuanto a los árboles de comportamiento que se enumeran a continuación. [38]

Diferencias:

- **Gestión por eventos**: Estos árboles para evitar hacer trabajo innecesario se evaluan mediante los eventos que pueden ser generados mediante triggers.
- **Las condiciones no son nodos hoja**: Aunque existe la posibilidad de implementar las condiciones de la misma forma que en los árboles 'estándar', la do-

cumentación de UE4 nos recomienda que en su aplicación lo hagamos mediante Decorators.

- **Nodos Paralelos**: Para simplificar comportamientos paralelos, UE4 nos ofrece el nodo 'Simple Parallel' que nos facilita esta tarea. Una posible utilidad para estos nodos es que tienen la capacidad de abortar un comportamiento, ya que se puede comprobar de manera simultanea si las condiciones de un comportamiento se siguen cumpliendo. En caso de que dejen de cumplirse pueden abortarse inmediatamente.
- **Nodo Servicios**: El nodo 'Services' nos permite asociar tareas que se repiten en intervalos de tiempo. Estas tareas pueden ser comprobar si fuera necesario dar prioridad a otro objetivo. Este nodo se puede asociar a otros

Esta documentación nos afirma que es más sencillo más fácil de depurar y más fácil de optimizar estos árboles de comportamiento gracias a las mencionadas diferencias.

2.2.3. Sistemas multi-agente y sistemas multi-agente auto-adaptativos (AMAS)

En esta sección se comentarán las propiedades y características de los sistemas multiagentes además de algunos de los variados desempeños que se les han dado en diversas aplicaciones interactivas o videojuegos.

2.2.3.1. Sistemas multi-agente

Los sistemas multi-agente son sistemas compuestos por entidades denominadas .ªgentes"que normalmente se simulan por odenador. Las principales características de los agentes es que tienen la capacidad de decidir por ellos mismos de forma autónoma y además de ser capaces de interactuar con otros agentes en función de las características de éstos. Al final, a partir de estas capacidades individuales de los agentes, mediante la interacción de estos agentes se puede resolver una gran amplitud de variedad de problemas[7].

Debido a que los sistemas multiagentes están basados en agentes que operan de forma independiente, una de las características principales de estos sistemas es que pueden ser utilizados para modelizar e implementar sistemas distribuidos de manera descentralizada. Los agentes son entidades que observan su entorno y lo que pueda haber en él y reaccionan según las normas que tienen estos agentes.[9]

2.2.3.2. Sistemas multi-agente auto adaptativos

Los sistemas multi-agente auto adaptivos surgen a partir de los sistemas multiagente debido necesidad de encontrar soluciones a problemas cada vez más complicados por la creciente complicación de las aplicaciones dia a día[1]. Debido a la evolución y sofisticación de estos sistemas a lo largo del tiempo, la necesidad de sistemas más robustos, más autónomos y más complejos que resuelven estos problemas crece a la par que estas tecnologías. Según la referencia mencionada, todos los sistemas basados en multiagentes son autoadaptativos en cierto grado. En su artículo defienden que este tipo de adaptatividad es pequeña y utilizan el término **strong adaptation** traducido como adaptación fuerte para referirse a la capacidad de un

sistema multiagente para realizar acciónes coherentes frente a sucesos imprevistos que pueden ocurrir en entornos cambiantes.

Para conseguir este propósito de auto adaptatividad, tal y como se explica en la metodología **ADELFE** existe la posibilidad de diseñar un sistema que en conjunto si es autoadaptativo, aunque los agentes que de manera individual no lo sean. El objetivo de la metodología ADELFE consiste en una forma de trabajar que pretende dar una forma de diseñar a quienes quieran diseñar un sistema multiagente colaborativo con la finalidad de que sea mas sencillo identicar los componentes del sistema para que este cumpla las características deseadas, ya que puede llegar a ser muy difícil encontrar una solución eficaz para sistemas complejos abarcando todos los posibles imprevistos que pueden surgir en un sistema cambiante.[1]

2.2.3.3. Sistemas multi-agente en aplicaciones interactivas

Podemos ver la gran flexibilidad que tienen los sistemas multiagentes para resolver problemas en el hecho de que se han resuelto multitud de ellos utilizando estas técnicas. Existen gran cantidad de artículos que hablan sobre este tipo de soluciones como la optimización de colonias de hormigas mediante la colocación de feromonas en el suelo que utilizan las hormigas compañeras como referencia para utilizar esos caminos[4], o la posibilidad de guiar grandes grupos [3] en áreas urbanas.

Por supuesto también está incluido su uso en los videojuegos como veremos en algunos artículos comentados brevemente a continuación. Podemos encontrar métodos para generación de ciudades incluída su red de carreteras[5].

Existen también implementaciones del conocido Conways Game of life basada en multiagentes que considera a cada celdilla del tablero como un agente individual que interactua con los ocho agentes que están colocados a su alrededor [10].

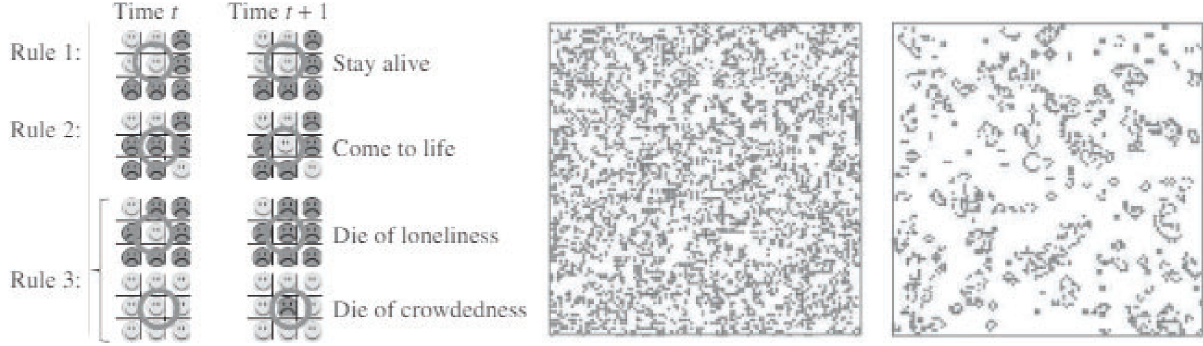

Figura 2.2: Ejemplo de 'Conways game of life' Imagen sacada de la referencia [10]

Como se puede ver, ya se han hecho desarrollos similares para propósitos distintos al desarrollado en la propuesta de este trabajo en unity3D. Más concretamente en este caso, vamos a hablar de un uso de multiagentes hecha en la versión 4 de Unity[11]. El citado artículo intenta mejorar los resultados de una simulación que pretende imitar el comportamiento del pathfinding de los pasajeros que buscan la ruta para llegar a una salida de un aeropuerto. Estos autores en su implementación comentan haber conseguido un rendimiento bastante elevado puesto que dicen conseguir simular hasta 600 agentes en un procesador i5 con 8 gigas de ram y una

tarjeta gráfica AMD Radeon HD 7800 a 60fps por segundo. Esta medida puede servirnos como referencia aproximada, ya que aunque por las fotografías incluidas en el artículo se ve que pretenden mantener la carga gráfica al mínimo, no comenta otros elementos que pueden influir en la carga del procesador como los materiales de unity utilizados. Tampoco indica el modelo concreto de cpu. Aunque por la época del artículo, quizás se pudiera suponer que se trata de un i5 de cuarta generación, ya que el artículo tiene fecha del 2014 y sería de los modelos más contemporáneos.

La solución propuesta para esta simulación consiste en unos agentes representados en forma de caja que van buscando unas señales. Cuando se aproximan lo suficiente a una señal, para simular su visibilidad se lanzan cinco raycasts desde la posición del agente a dicha señal. Estos raycasts se lanzan uno al centro de la señal, y otros cuatro a sus respectivas esquinas. En función de la cantidad de los raycasts que lleguen satisfactoriamente a tocar la señal y su ángulo de incidencia se estima una visibilidad de la señal entre 0 y 1. Si la visibilidad es próxima a 0 el agente que la está viendo puede malinterpretar la señal e ir en una dirección incorrecta. Si la visibilidad es buena la dirección de la señal se extrapola dándo al agente una ruta hacia un waypoint. Si un agente no es capaz de ver una señal, se le asigna una ruta aleatoria dentro del nuevo waypoint hasta que sea capaz de ver una señal. Estas señales y waypoints deben ser colocadas a mano para cada escenario.

2.2.4. Matrices de interacción basadas en reglas

Como veremos a continuación, las matrices de interacción almacenan en sus respectivas celdas los diferentes comportamientos que deben ejecutar los agentes al interactuar con el resto de los agentes del entorno. Estas matrices pueden almacenar en cada celda varias reglas diferentes definidas como funciones que pueden discernirse en función de la prioridad de unas acciones, parámetros de entrada de las funciones, o estados en los que se encuentren los agentes. En la figura 2.3 podemos ver un ejemplo sencillo con agentes representando ovejas,cabra, hierba y lobos. La primera columna de la matriz representa las interacciones que el agente hace por sí mismo. La hierba crece, la oveja el lobo y la cabra se mueven. Las siguientes columnas representan las interacciones que pueden darse entre los diferentes agentes, y las condiciones que se tienen que dar para que se cumplan las siguientes acciones. En este caso la única condición que se tiene que cumplir para que los agentes puedan efectuar su interacción es la distancia 'D' que indica la distancia mínima. En caso de 'EAT'para que una oveja pueda comerse la hierba la distancia con la hierba tiene que ser de cero. En el caso de 'Procreate' la distancia tiene que ser de un metro. Y en el caso del lobo para que se pueda comer a una oveja o a una cabra la distancia tiene que ser de tres.

Source \ Target	∅	Grass	Sheep	Goat	Wolf
Grass	(Grow)				
Sheep	(Move)	(Eat, d = 0)	(Procreate, d = 1)		
Goat	(Move)	(Eat, d = 0)		(Procreate, d = 1)	
Wolf	(Move)		(Eat, d = 3)	(Eat, d = 3)	(Procreate, d = 1)

Figura 2.3: Imagen de ejemplo de una matriz de interacción sacada de la referencia [24]

Matrices de interacción:

- **Raw Interaction Matrix**: Esta matriz almacena en las celdas todas las posibles interacciones que pueden tener los agentes además de las condiciones necesarias para que puedan llevarse a cabo.
- **Refined Interaction Matrix**: Esta matriz tiene como añadido que las interacciones tienen un número entero indicando su prioridad. En caso de que dos acciones no puedan hacerse de forma simultánea, primero se escogerá aquella que tenga la prioridad mas alta.

2.2.4.1. IODA

IODA cuyas siglas son 'Interaction-Oriented Design of Agent simulations'. Esta metodología considera que todas las acciones de un agente son parte de una interacción ya sea con el entorno u otro agente. De este modo, todos los agentes de la simulación tienen la capacidad de interactuar de algún modo. Una característica importante sobre esta metodología es la separación de los agentes y sus interacciones. Esto permite la posibilidad de poder reutilizar las interacciones en otras simulaciones con desempeños parecidos o iguales. Es decir, en campos de aplicación similares.[24]

Se puede decir de forma resumida que el metodo IODA consta de las siguientes partes o características de las que un diseño basado en multiagentes puede beneficiarse:

- **Environment:** Es el entorno que está habitado por nuestros agentes y entidades. El entorno puede definir métricas no euclídeas que sean relevantes para la simulación. Como por ejemplo distancia de estatus social o grado de consanguinidad.
- **Interaction:** Las interacciones son las acciones que efectúan los agentes entre sí. Estas interacciones involucran una cantidad de agentes fija y tienen definidas cómo y bajo qué condiciones se ejecutan.
- **Interactions polymorphism:** El polimorfismo de interacciones. Dos agentes pueden ejecutar la misma acción pero lo hacen de forma distinta. En el artículo mencionado, se ejemplifica diciendo que un lobo y un humano comen pero son agentes distintos. Por lo que diferentes agentes pueden ejecutar las mismas interacciones.
- **Interaction typology:** Es posible que dos agentes no interactúen efectuando el mismo papel. Además también es posible que una misma interacción con otro agente genere resultados distintos.
- **Formal definition of an Interaction:** IODA describe de manera formal las interacciones en forma de tupla.
- **Interaction matrix:** La matriz de interacción que define las posibles acciones de un agente puede iniciar frente a otro, y aquellas que no requieren otro agente para llevarse a cabo.
- **Agent family:** Los agentes pueden pertenecer a familias que pueden compartir interacciones.
- **Agent Inheritance:** Gracias a la herencia entre agentes, otro agente puede interactuar con una familia de agentes con la misma interacción. El lobo y la

oveja pueden pertenecer a una familia de animales, y un cazador puede cazar animales.

- **Representacion del tiempo:** Símplemente la posibilidad de representar el paso del tiempo de forma precisa.
- **Reactive interaction selection:** Cada elemento de la matriz tiene un número que representa la prioridad. Esto permite la selección de interacciones en caso de ambigüedad.

2.2.4.2. The Galaxian Project

The Galaxian Project es un proyecto en el que se utiliza el método IODA para el desarrollo de una inteligencia artificial que controlaría los agentes en un videojuego de naves espaciales. Este proyecto pretende demostrar las capacidades que tiene esta metodología en entornos interactivos tal y como lo son los videojuegos. Además, el uso de estas interacciones entre agentes mediante método IODA permite que se añadan una cantidad infinita de extensiones a la inteligencia artificial. [26]

En el documento mencionado se comenta como **The Galaxian Project** que inicialmente fué programado en java, se convierte al lenguaje c# para poder ser utilizado en un motor de videojuegos comercial como lo es Unity 3D. **The Galaxian Project** muestra como se pueden utilizar las matrices de interacción para definir el comportamiento de distintos agentes. En este caso de diferentes naves espaciales en diferentes equipos. En la siguiente figura 2.4 podemos ver la matriz de interacción usada en Galaxian.

Sources \ Targets	∅	BFighter	WFighter	BFrigate	WSquad
BFighter	(SeekTarget; 0) (Fire; 8) (Explode; 9)		(Confront; 2; 1000) (Intercept; 3; 1000) (Engage; 4; 1000) (Escape; 5; 1000)		
WFighter	(SeekTarget; 0) (Fire; 8) (Explode; 9)	(Confront; 2; 1000) (Intercept; 3; 1000) (Engage; 4; 1000) (Escape; 5; 1000)	(CreateSquad; 2; 30)		(Join; 6; 100) (Follow; 7; 1000)
BFrigate					(FightBack; 0; 200)
BCruser	(LaunchFighter; 0)				
WCruser	(LaunchFighter; 0)				
WSquad	(Disband; 0) (Fire; 3)			(Engage; 1; 5000)	(Merge; 5; 50)

Figura 2.4: Matriz de Galaxian. Imagen tomada de la referencia [26]

.

2.2.4.3. FormatStore

Format Store es un proyecto muy interesante ya que se trata de un **Serious Game**. La intención de este proyecto es permitir a estudiantes de escuelas de negocios que puedan aprender y practicar en un entorno interactivo virtual las habilidades requeridas en una tienda de cara al público. El proyecto dispone de varios niveles y de varios temas para entrenar estas habilidades como lo son el trato con los clientes

o la gestión de la tienda. Este proyecto pretende proponer un nuevo modelo educativo innovador en comparación al sistema de enseñanza clásico. Este no es el único proyecto enfocado hacia el aprendizaje mediante entornos interactivos virtuales. Ya se han hecho diversos serious games en los que se pretende enseñar habilidades de formas innovadoras a gente que quiere mejorar sus conocimientos en diferentes ámbitos.[25] En la figura 2.5 podemos ver el aspecto que tiene Format Store.

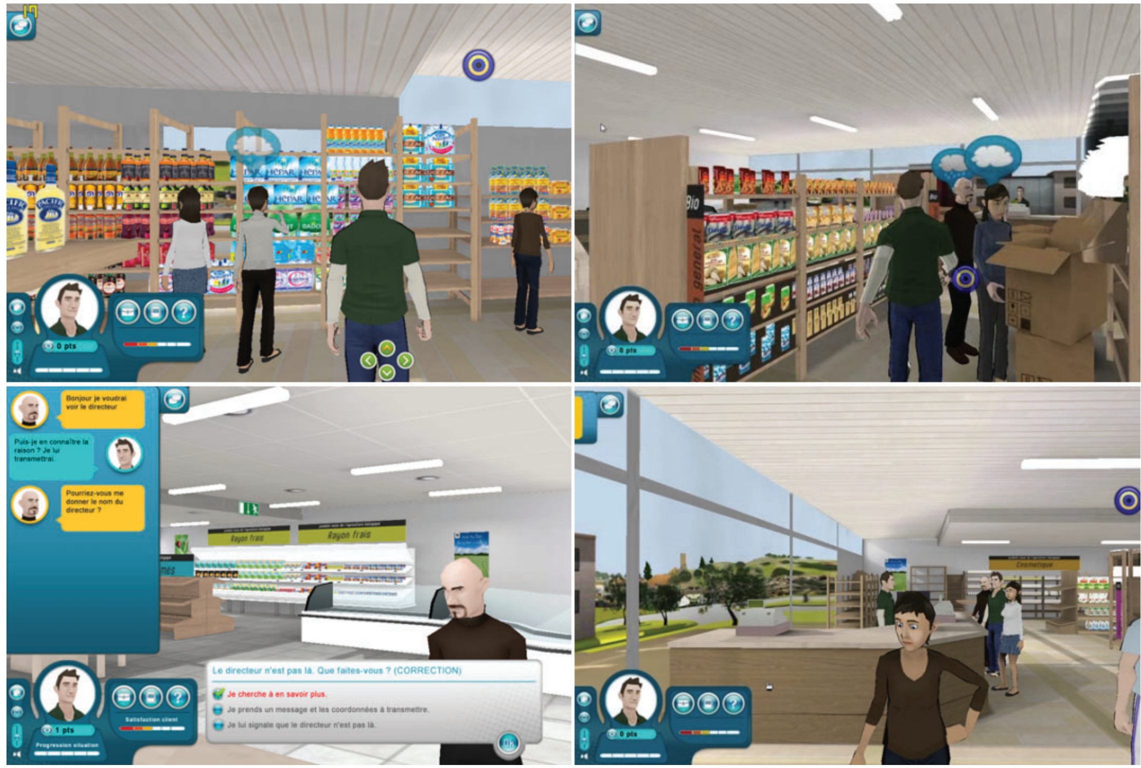

Figura 2.5: Imágenes de Format Store. Imagen tomada de la referencia [25]
.

En Format Store al igual que en The Galaxian Project se utiliza el método IODA. Los sistemas multiagentes han resultado ser bastante efectivos a la hora de modelar comportamientos colectivos mediante las interacciones que se ejecutan entre ellos mismos.

Una diferencia importante frente a The galaxian project' es que Format Store incluye un agente controlado por una persona. El jugador posee una interfaz mediante la cual puede elegir las respuestas que dar a los clientes que son simulados por el sistema multiagente. En función del rendimiento del usuario, existe un controlador de juego que analiza las acciones que está realizando el jugador. Dependiendo de si si está tomando las decisiones adecuadas y respondiendo correctamente a las preguntas de los clientes o está respondiendo incorrectamente, este controlador de juego le asignará una buena o una mala puntuación. Adicionalmente es posible aumentar o disminuir la dificultad del videojuego basandose en que la cantidad máxima de agentes que el controlador de juego puede poner en el entorno es configurable.

Una gran propiedad de Format Store es su escalabilidad. Tal y como el artículo explic, esta capacidad fué pensada con especial énfasis para poder satisfacer que las necesidades educativas futuras puedan añadirse manteniendo así el proyecto

actualizado. Un ejemplo de esto es que los agentes pueden tener diferentes scripts con las posibles converscioness con diferentes comportamientos tipo que se encuentran en los clientes que cubrirían distintos ámbitos pedagógicos.

La auto adaptatividad del sistema permite que pueda haber productos de la tienda que sean quitados o que un producto nuevo sea añadido para que un agente pueda preguntarle al usuario algo relacionado con el producto. También existe la posibilidad de colocar mal señales para que los agentes se confundan y vayan a preguntar al jugador sobre si ese producto se encuentra en la tienda ya que no lo han encontrado.

Las conversaciones que pueden tener los diferentes agentes están escritas a mano en un fichero XML para que la aplicación pueda extraer las conversaciones y las posibles respuestas de ese fichero. De esta forma es relativamente sencillo añadir más situaciones que puedan darse en una tienda en caso de que se requieran.

En definitiva, los autores de este artículo planean utilizar este sistema que ha dado buenos resultados en la recreación de un entorno realista de una tienda virtual y extenderlo a supermercados grandes. También barajan la posibilidad de emplearlo en escenarios de índole similar pero que abarquen otros ámbitos.

2.2.5. Algoritmo HPA*

En este apartado se va a proceder a comentar el artículo del algoritmo HPA* cuyas siglas corresponden a "Near optimal Hierarchical path-finding A*"de los autores que son: Adi Botea, Martin Müller y Jonathan Schaeffer. Cuya motivación fué encontrar la ruta de menor distancia viajando en coche desde la ciudad de Los Ángeles hasta Toronto [20]

El algoritmo HPA* que han desarrollado estos autores mencionados en el párrafo anterior, propone una solución para el problema del path-finding en videojuegos comerciales que normalmente tiene que ser resuelto en tiempo real basado en mapas de rejilla. En éste artículo se comenta que el algoritmo A* tiene el problema de que el coste de cómputo de este algoritmo se incrementa con el tamaño del espacio de búsqueda, pudiendo darse el caso de que en sistemas con restricciones de hardware como pueden ser de memoria de procesador, supongan una limitación que pueda no cumplir los requisitos de la aplicación [20]. Según los estudios y pruebas hechos en este artículo a base de experimentar el algoritmo en videojuegos como Baldur's Gate éste método muestra una gran mejora velocidad de cómputo de hasta diez veces más rápido que A*, con la pequeña pega de que las rutas encontradas son un 1 % menos óptimas que las de A*.

Para conseguir este gran incremento en la velocidad del pathfinding, tal y como se ve en la figura el algoritmo divide el espacio de búsqueda en subfragmentos más pequeños que son conexos entre si organizados en diferentes niveles de jerarquías. El algoritmo permite jerarquizar el problema en tantos niveles como se crean necesarios. Para ello se crea una simplificación del espacio de búsqueda que permite encontrar al algoritmo A* una ruta aproximada por los subconjuntos por los que debería ir la ruta óptima[20].

En la figura 2.6 se puede ver como el algoritmo divide el espacio de búsqueda. En este caso es una rejilla de 40x40 en rejillas más pequeñas de 10x10. A su vez, la

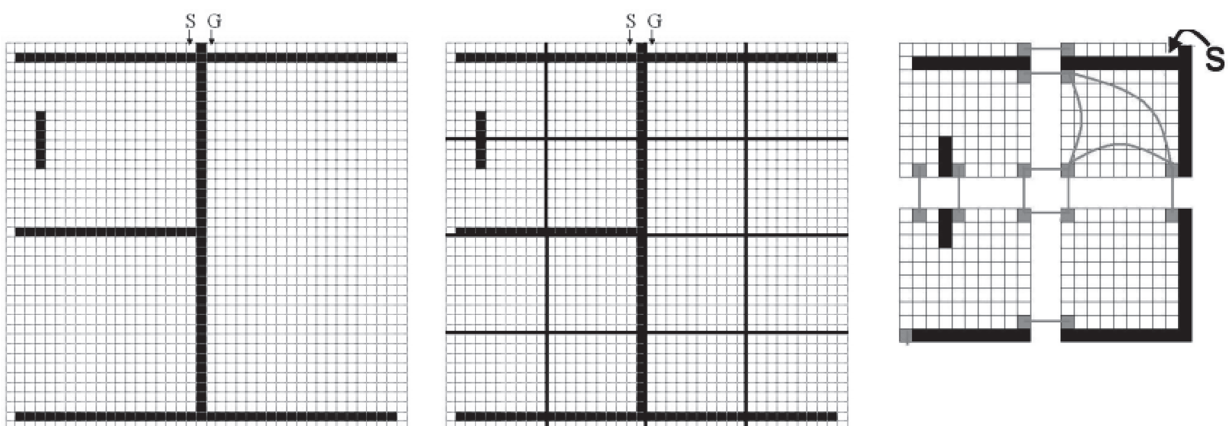

Figura 2.6: Representación de la abstracción de una rejilla por el algoritmo HPA*. Imagen tomada de la referencia [20]

.

conectividad entre si de estas rejillas más pequeñas se representa de manera simplificada con nodos colocados en las esquinas de las rejillas pequeñas. De esta forma, se hace un grafo que representa de forma general el mapa inicial, simplificando enormemente el espacio de búsqueda.

Esta simplificación del espacio de búsqueda puede ocasionar que la ruta encontrada no sea óptima. Para esto, tras haber encontrado una posible ruta se aplican métodos de suavizado de la ruta encontrada que intenta optimizar un poco más el primer resultado obtenido. La solución que propone este artículo y que según se comenta da resultados bastante buenos, es siempre que se pueda, sustituir rutas subóptimas por líneas rectas.

2.2.6. Malla de Navegación o NavMesh

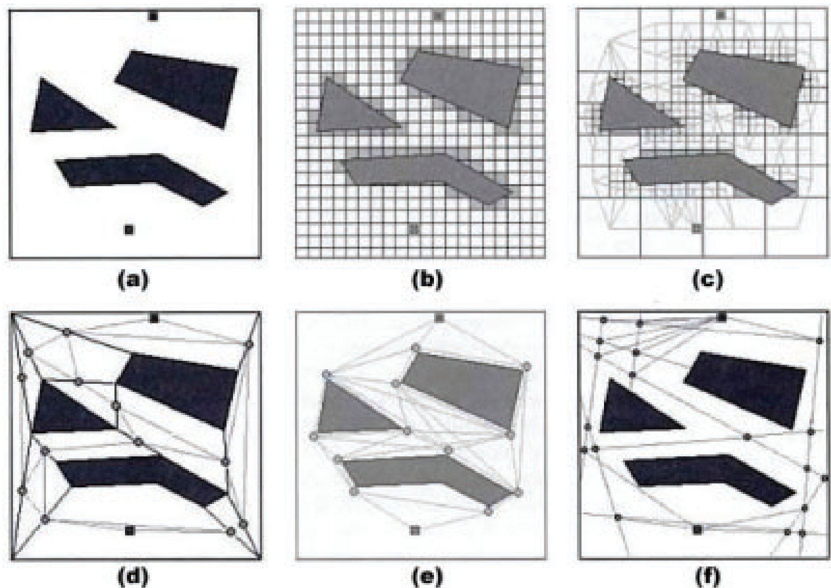

Figura 2.7: Diferentes posibles representaciones con nodos de una malla de navegación. Imagen de la referencia [19]

La malla de navegación o NavMesh es una herramienta muy utilizada y extendida para solucionar los problemas que nos presenta el pathfinding. Los NavMesh representan de una forma más exacta la superficie navegable que los métodos basados en rejilla. Además, como las áreas de los polígonos que conforman el NavMesh abarcan el área de varios elementos que ocuparían una rejilla, el número de nodos necesarios para representar la búsqueda se reduce considerablemente [18]. Como podemos ver a continuación en los siguientes apartados de este documento se procederá a analizarlos, y veremos que esta herramienta de pathfinding se encuentra disponinble entre las principales herramientas de desarrollo de videojuegos que actualmente se encuentran disponibles a cualquier usuario.

La malla de navegación es una malla poligonal compuesta a base de un conjunto de triángulos representados por una matriz de vértices conjunto con otra matriz de tripletes de índices que nos indica qué vértices forman cada triángulo. De esta manera, el conjunto de triángulos representan la superficie por donde se pueden mover los agentes. Los triángulos de las mallas pueden llevar asociados pesos al igual que un árbol. Lo cual permite a la malla hacer una representación más adecuada al entorno y permite al algoritmo de pathfinding en caso de Unity A* [27] en caso de UE4 el algoritmo HPA* [20] encontrar un camino más adecuado a su propósito. En el caso del algoritmo HPA* hay que mencionar que no encuentra el camino óptimo si no que intenta encontrar un camino bastante óptimo sin garantizar la mejor ruta.

Los NavMesh se calculan en tiempo de producción de forma automática en función de los parámetros que nos brinda cada herramienta. Aunque el concepto aplicado es el mismo para todas las herramientas, cada herramienta debido a sus particularidades no lo implementa de manera exactamente igual habiendo pequeñas diferencias en cada una de ellas. Aunque estas particularidades las veremos en detalle en cada una de las secciones referidas a cada herramienta en si.

El NavMesh es bastante eficaz para muchos propósitos ya que responde a las necesidades comerciales actuales y tiene un coste razonable a la hora de calcular rutas, el NavMesh es de carácter estático y aunque según cada herramienta tiene utilidades para paliar los problemas debido a sus cárácteristicas, en general no tiene en cuenta posibles obstáculos que puedan modificar el entorno en tiempo de ejecución. Su concepto como base supone ciertas limitaciones que se comentarán en cada apartado según la herramienta en particular y en general tampoco no nos brinda funcionalidades que nos den la capacidad de interactuar con otros agentes.

2.2.6.1. NavMesh en Unity

En este apartado del documento se procedederá a hablar de las particularidades del funcionamiento del NavMesh en Unity 3D.

Para su estudio se ha procedido a diseñar un pequeño entorno de prueba para así ver las características del programa de primera mano. En este pequeño entorno se han añadido una rampa, escaleras, un pequeño laberinto y un arco con el fin de comprobar si los agentes pueden pasar por debajo de él.

Figura 2.8: Ejemplo de NavMesh. Imagen tomada de Unity

En la figura se pueden observar las áreas de distintos colores que representan las zonas transitables por los agentes. Los colores nos ayudan a ver de manera sencilla el coste de tránsito por las diferentes zonas del escenario. A continuación, en este documento se detallará el proceso necesario para construir una malla de navegación como la que podemos observar en ejemplo mostrado en la figura 2.2. Además se analizarán y comentarán las diferentes opciones que nos ofrece esta herramienta para la generacion y uso de estas mallas de navegación.

2.2.6.2. Áreas del NavMesh en Unity

Unity nos permite asignar el coste de tránsito de hasta 32 áreas distintas. Si bien 32 puede ser suficiente para un gran número de aplicaciones, si las particularidades de un proyecto requieren más tipos de áreas distintas entonces esta limitación quizás impida el desempeño de ese experimento en cuestión.

Como podemos ver en la figura 2.9 , algunas de las áreas están reservadas por defecto. Éstas áreas están remarcadas en gris como no editables y llamadas "Built-in 0,1 y 2".Estas etiquetas no editables son para los siguientes propósitos de Unity. La zona por defecto "Walkable", para áreas normales sobre las que los agentes pueden transitar, o por el contrario "Not Walkable". Esta última zona símplemente a la hora de computarse la malla de navegación tan solo actuará como si no existiera y no se generará geometría del NavMesh en esa zona. Finalmente el área "Jump", ha sido reservada para nodos complementarios que representan zonas que pueden estar desconectadas pero lo suficientemente cercanas como para que un agente puede saltar entre una y otra.

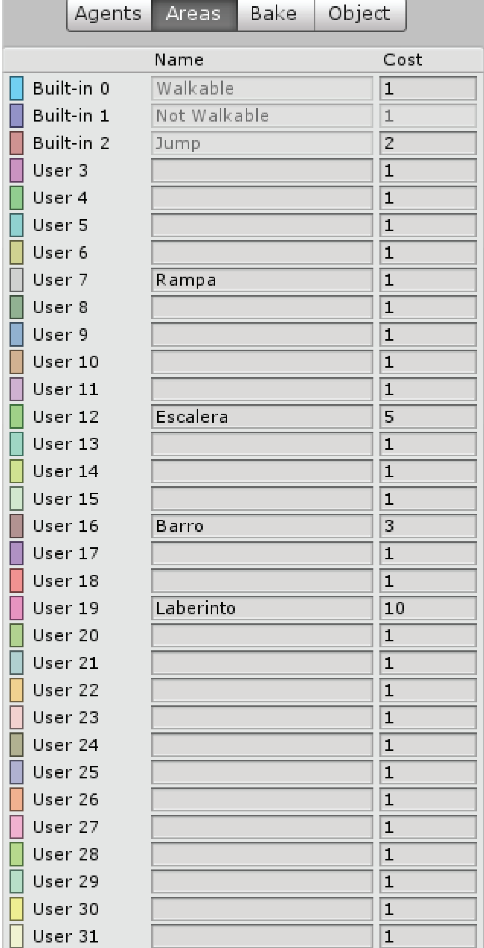

Figura 2.9: Areas de Unity

En nuestro caso, tan sólo necesitaríamos un tipo de área, ya que toda la superficie sobre la que se distribuirán nuestros agentes, aunque estará llena de obstáculos que pueden alterar el espacio sobre el que pueden transitar los agentes, sería una superficie plana y uniforme.

La asignación de las áreas del NavMesh puede hacerse de forma manual por cada objeto deseado del escenario. Esto puede ser una árdua tarea para las manos de los usuarios si el escenario tiene una gran cantidad de objetos o el escenario es de un tamaño muy grande. De cualquier forma, Unity 3D gracias al componente NavMeshSurface [30] nos permite automatizar el proceso mediante el uso de scripts.

2.2.6.3. Agentes del NavMesh en Unity

En la figura 2.10 podemos observar los diferentes parámetros que son configurables en Unity 3D para utilizarlos después en nuestro entorno tridimensional. En principio unity nos permite crear tanta cantidad de agentes distintos como sean necesarios.

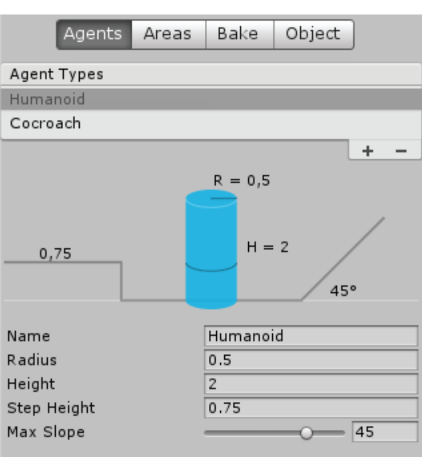

Figura 2.10: Imagen del configurador de agentes tomada del programa Unity

Propiedades:

- **Name**: El nombre mediante el cual vamos a diferenciar nuestro agente en caso de tener varios distintos
- **Radius**: El radio que abulta nuestro agente. Gracias a esto podemos hacer que un agente pueda o no pasar por un hueco debido a su tamaño.
- **Height**: La altura del agente. Para impedir el paso a agentes que sean demasiado altos para un pasadizo demasiado bajo.
- **Step Height**: La altura máxima del escalón que puede subir un agente en caso de encontrarse con uno.
- **Max Slope**: Pendiente máxima de una rampa por la que el agente puede subir o bajar.

Hay más atributos que se pueden configurar. Los atributos del agente en si se pueden encontrar en el componente NavMeshAgent. [29] Entre estos atributos están la velocidad del agente o también su aceleración. Las distintas áreas por las que se le permite transitar a un agente, la calidad de la capacidad del agente para esquivar obstáculos con el fin de aliviar la carga de procesador, pudiendo ser alta o ninguna, dejando así tan sólo la simulación de colisiones con otros agentes.

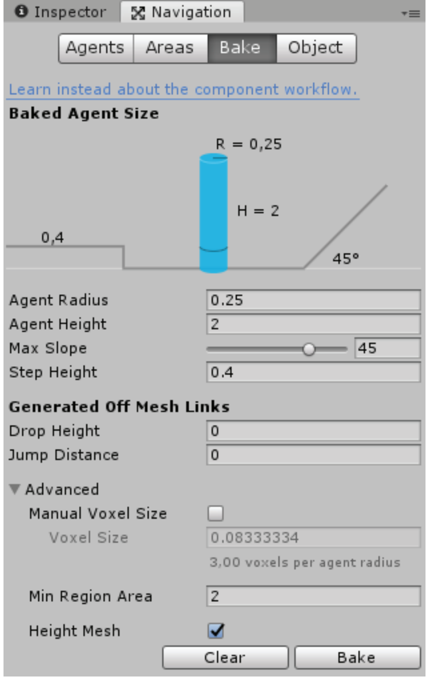

Figura 2.11: Imagen de las propiedades del generador de NavMesh

Como se puede ver en la figura 2.11 se pueden ajustar los parámetros para la creación del NavMesh. Se puede configurar el radio del agente para no generar malla en aquellos lugares donde los agentes no quepan y de por si no puedan entrar. Lo mismo con su altura y tamaño de escalón máximo. También se puede ajustar la distancia máxima de salto entre dos zonas que no estén conectadas.

En la pestaña **Advanced** también nos permite seleccionar el tamaño de los voxels que representarían la rejilla que se va a utilizar para generar la malla de navegación. A menor tamaño, mayor consumo de memoria y de cpu, pero obtendremos una mejor aproximación de

la malla al escenario real. Al contrario que en unity que tan solo tiene la capacidad de hasta 32 posibles áreas distintas, en Unreal Engine 4 podemos definir tantas como queramos. Además de una funcionalidad adicional.

Adicionalmente, aunque el generador de NavMesh de unity es capaz de generar automáticamente saltos entre zonas inconexas de la malla, en caso de que necesitemos definir estas conexiones manualmente, tenemos la posibilidad de usar el componente **OffMeshLink** [31]. Este componente permite al usuario unir de forma manual regiones no conexas que podamos tener en el entorno para que el pathfinding sea capaz de considerar estos caminos como posibles rutas. En general, este componente nos permite representar elementos por los que se pueda transitar de un sitio a otro que pero que no sean representables por la malla de navegación debido a sus características.

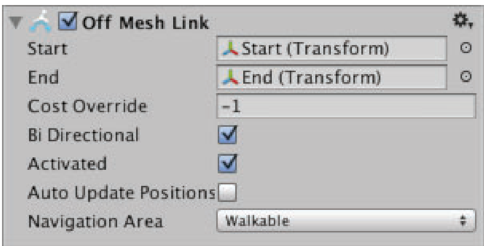

Figura 2.12: Figura sacada de la documentación de unity.

2.2.6.4. NavMesh en Unreal Engine 4

Como podemos ver en la figura 2.13 Unreal Engine 4 tiene propiedades parecidas a las de Unity3D. Existen algunas diferencias como la capacidad para elegir el ancho y el alto de las celdas que generan el NavMesh. Por ejemplo la propiedad **Cell Size** que sería el tamaño horizontal de la rejilla utilizada para generar el navmesh, o **Cell Height** que permite al usuario elegir la altura de la celda. Tenemos algunas propiedades como lo son **Max Slope** y **Max Step Height** que serían completamente análogas a las de Unity 3D.

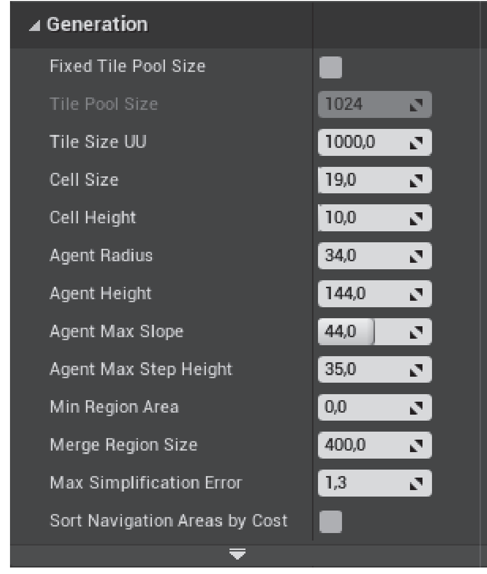

Figura 2.13: Propiedades del generador de NavMesh en UE4.

Además, existen algunas adicionales en este motor. Como lo son **Max Simplification Error** que permite ajustar la simplificación automática del NavMesh para obtener una geometría mas simple y **Min Region Area** que en caso de generar un área más pequeña que la indicada en este campo será descartada.

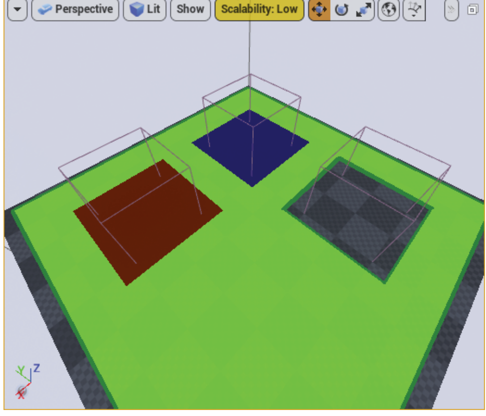

Figura 2.14: Imagen del NavModifierVolume.

Mientras que en Unity como hemos visto en el apartado anterior, la manera de indicar el coste de tránsito de las zonas es asignando la geometría de esa zona a un área en concreto, en Unreal Engine 4 tenemos la herramienta **NavModifierVolume** que consiste cumple análogamente la misma función con la diferencia de que el usuario deberá colocar unas cajas del tamaño y forma en aquellas zonas que desee que modifican el coste de esa zona [36]. Al igual que en unity tenemos las siguientes áreas ya definidas: **NavArea_Default**, que representaría el area transitable por defecto; **NavArea_Null**, zona por la cual no se genera NavMesh; **NavArea_Low**, zona por defecto de coste bajo; y **NavArea_Obstacle**, zona por defecto de alto coste que representaría una zona de obstáculos o de difícil tránsito.

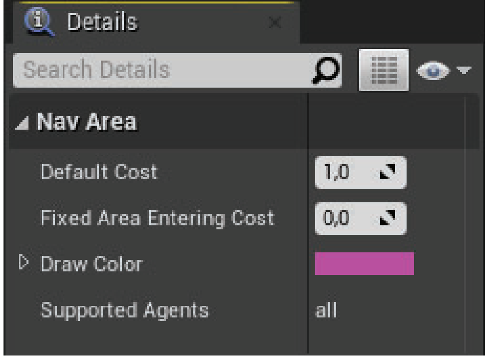

Figura 2.15: Imagen de las propiedades de coste de un area.

Como podemos ver en la figura 2.15, Unreal Engine 4 da al usuario un poco más de libertad a la hora de definir el espacio por el cual pueden transitar los agentes. A parte de poder definir el coste de tránsito por el área, es posible definir además el coste de entrar en esa área. Esto nos permite simular eventos que pueden ocurrir en un tránsito. Por ejemplo si para entrar a determinada zona un agente tuviera que cambiar de estado para poder entrar a esa zona pero luego una vez dentro el coste de transitar por dentro de

esa área fuese igual al de haber caminado por la
anterior.

De una forma similar a la que funciona el OffMeshLink en Unity, tal y como se puede ver en su documentación Unreal Engine 4 nos ofrece los NavLinkProxy [37]. Que permiten de igual forma unir zonas donde donde la malla de navegación no puede generarse debido a las características del entorno. Estos NavLinkProxy permiten indicar a los agentes rutas adicionales deseadas por el usuario del programa.

2.2.6.5. NavMesh en Source Engine

Valve Software, desde sus inicios, junto con sus videojuegos siempre ha proporcionado a sus usuarios de forma gratuita tanto su editor de niveles llamado Valve Hammer Editor, como el código fuente de sus juegos. De esta forma sus usuarios siempre han podido crear su propio contenido. Hay que mencionar que estas herramientas están centradas principalmente en las necesidades de desarrollo de la compañía más que en el uso de propósito general que si tienen Unreal Engine 4 o Unity 3D.

Si bien el NavMesh de Source Engine cumple los propósitos necesarios para el correcto funcionamiento de la inteligencia artificial de los videojuegos hechos por la compañía, por lo que podemos encontrar, aparentemente no podemos añadir tipos nuevos de áreas por lo que solo existen tres áreas distintas: **Crouch Area**, área por la que sus agentes de inteligencia artificial pueden transitar al ir agachados, **Jump Area**, área por la que los agentes deben saltar para que esta pueda ser transitada; y **Navigation Area**, el área transitable por defecto.

Como podemos ver en su documentación oficial, Source Engine ofrece un generador automático de mallas de navegación [39]. En caso de que el generador no cumpla con las expectativas del usuario, el motor nos ofrece la posibilidad de editar la malla de navegación manualmente a partir de las siguientes operaciones.

Operaciones:

- **NewArea:** Define el contorno rectangular del área del navmesh.
- **Split:** Parte un área en dos distintas.
- **Merge:** Combina dos áreas en una sola.
- **Splice:** Empalma dos áreas separadas creando un puente entre ellas.

Además de las mencionadas operaciones que nos permiten modificar el NavMesh, existen opciones adicionales que dan soporte a la inteligencia artificial de sus juegos. Entre estas opciones hay la posibilidad de asignar diferentes etiquetas que permiten a la inteligencia artificial saber si una zona es adecuada para realizar ciertas acciones como puede tener una zona donde esconderse, o la posibilidad de unir regiones del NavMesh mediante conexiones unidireccionales o bidireccionales. A parte, también se incluyen herramientas que añade información adicional en el NavMesh que permite a los agentes controlados por inteligencia artificial utilizar elementos del entorno tridimensional como las escaleras de mano.

Capítulo 3

Propuesta

3.1. Desarrollo de la inteligencia artificial de Executanks

Tras conocer las posibilidades que ofrecen actualmente las herramientas de creación de videojuegos que se encuentran disponibles, y tras haber analizado artículos que proponen soluciones a este tipo de problemas, como puede serlo la dependencia de los algoritmos basados en A* en estructuras de datos pregeneradas como las mallas de navegación que pueden provocar altos costes de procesamiento sobre todo para espacios grandes de búsqueda que pueden generar pequeños retrasos en una ejecución en tiempo real[6], se seleccionaron y se adaptaron las técnicas que tendían a cumplir los requisitos necesarios para la creación de la inteligencia artificial que se requería. Muchas de las técnicas y de los métodos mencionados anteriormente han sido descartados debido a la dificultad de adaptarlos o aplicarlos en este particular ámbito, o por que no se ajustaban a las características de este proyecto. Por ello, las razones mencionadas en los requisitos sugerían el uso de un sistema multiagente para el desarrollo de este trabajo.

3.1.1. Requisitos

Los requisitos responden al diseño y características de interactividad del juego. Además de la necesidad comercial de hacer que éste pueda funcionar en los dispositivos más baratos posibles con la finalidad de poder abarcar el mayor abanico de mercado posible.

Requisitos:

- El pathfinding debe conseguirse sin rejilla.
- La IA debe adaptarse a un entorno que cambia en tiempo de ejecución.
- El tiempo de ejecución tiene que ser viable en dispositivos de bajo consumo. Como lo son los teléfonos móviles.
- Debe interactuar correctamente con otros agentes para no colisionar con ellos sin que esto afecte al pathfinding.
- Debe gestionar la dirección e inercia del agente tanque.

- Debe poder diferenciar entre diferentes obstáculos para actuar en consecuencia según el tipo de obstáculo.
- Debe poder escalar la cantidad de agentes a un coste bajo para que no se ralentice la velocidad de ejecución.
- Debe poder continuar comportándose correctamente tras encontrarse con grandes cantidades de obstáculos como escombros.
- Debe poder continuar comportándose correctamente tras encontrarse con grandes cantidades de obstáculos como escombros.
- Debe poder recuperar el comportamiento normal tras una colisión. Los agentes no colisionan entre ellos. Pero puede ser que el jugador los choque o haya escombros que desvíen su trayectoria.

Por estos requisitos se tomó la decisión de implementar un sistema multiagentes auto adaptativo basado sin rejilla. En el caso de Unity[27] y Unreal Engine 4 que utilizan A* [35], o a pesar de que se usase HPA* que incluso llega a ser hasta diez veces mas rápido que A* en su propio artículo [20] mencionan que el algoritmo funciona bien con una cantidad de obstáculos relativamente baja. Además de que calcular las modificaciones de la rejilla que requieren los algoritmos como HPA* o A* en un entorno de ejecución tan cambiante y con tanta cantidad de obstáculos como pueden encontrarse en Executanks suponía un coste extra que simplemente puede ser descartado al utilizar un sistema multiagente.

Además como podemos ver en la documentación de Unity[28] aunque está diseñado para responder a muchas necesidades de los usuarios, carece de algunas características requeridas para este proyecto en particular como la necesidad de funcionar sin NavMesh ni tener atributos como masa para responder a interacciones como posibles colisiones.

3.1.2. Entorno

El entorno en el que se van a hacer las pruebas es un suelo plano cuadrado que representaría un campo de batalla de 16 kilómetros cuadrados. Durante la ejecución de las pruebas, aparecen grupos de agentes sobre esa superficie a lo largo del tiempo. El entorno es cambiante teniendo una cantidad variable de obstáculos que tienen forma convexa. Existen tres tipos de obstáculos con forma distinta, existiendo la posibilidad de que estos sean eliminados en tiempo real por el jugador. Por otra parte, el jugador tiene limitados sus movimientos a un radio de 1250 metros desde el centro del cuadrilátero. Esto está hecho con la finalidad de que los agentes puedan aparecer siempre lo suficientemente lejos como para que el jugador no llegue a apreciar como los agentes van apareciendo durante la ejecución del juego.

Tipos de obstáculos:

- Cuadrado:
- Rectángular:
- Circular:

Con la finalidad de evitar de que dos obstáculos de este tipo se junten y formen una figura cóncava propensa a crear una situación de bloqueo de los agentes,

los obstáculos se colocan de manera aleatoria con una separación mínima de 128 metros entre sus centros. Para evitar que los agentes aparezcan en el interior de los obstáculos al principio de la ejecución se cuadrifica el escenario en forma de árbol bipartido reservando la mitad de las regiones para obstáculos y la otra mitad como espacio libre para hacer aparecer agentes. En caso de que el jugador esté cerca de los bordes del entorno circular, como se ve en la figura 3.1 existe un margen vacío para que los agentes siempre puedan aparecer a una distancia suficientemente lejana del jugador.

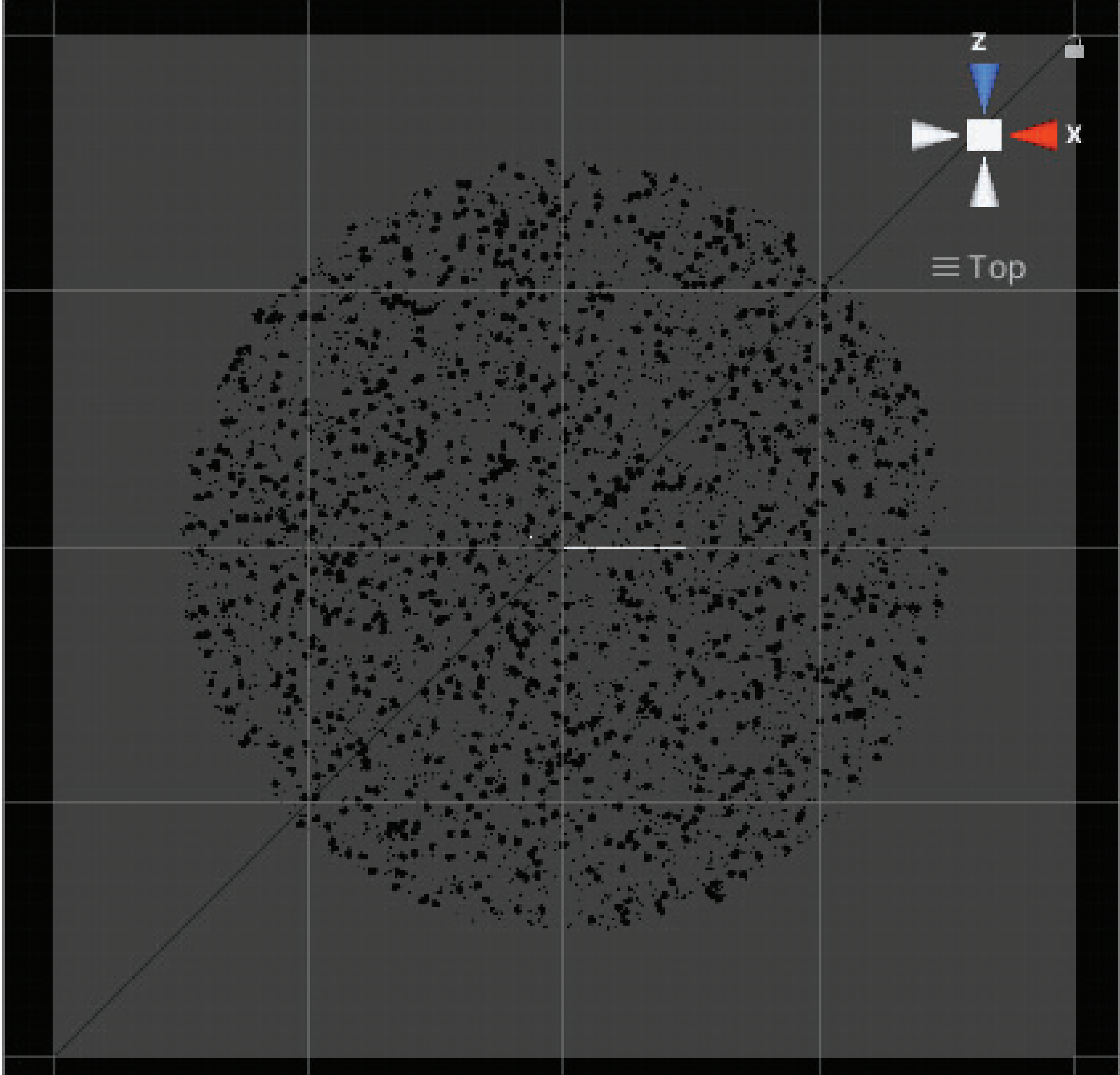

Figura 3.1: Vista cenital del entorno.

3.1.3. Descripción de los Agentes utilizados

En este proyecto se desarrollaron tres agentes distintos. Estos tres agentes son tres tanques cada uno con diferente armamento que los hace distintos a los ojos del jugador, pero que son iguales desde el punto de vista del comportamiento basado

en multiagentes. Esto es debido a que los tres agentes heredan de una misma clase que es la que contiene el código que implementa el comportamiento de estos agentes. Por lo tanto comparten exactamente el mismo código y las mismas normas, y como consecuencia se genera el mismo comportamiento para todos.

Las dificultades que nos podemos encontrar a la hora de controlar por inteligencia artificial vehículos pueden llegar a ser realmente complicadas. Los elementos como las marchas, los embragues, en algunos casos incluso el combustible restante son variables que han de ser consideradas para tomar decisiones óptimas a la hora de manejar este tipo de agentes[12]. Aunque la inteligencia artificial propuesta en este trabajo presenta algunos de estos problemas al ser una inteligencia artificial orientada a vehículos que son tanques, el requisito fundamental que se deseaba conseguir era el mayor rendimiento posible para poder gestionar la mayor cantidad de tanques posibles en un juego. De este modo, esta clase de elementos mencionados como el combustible o las marchas de los vehículos están simplificados al máximo con la finalidad de que la inteligencia artificial no necesite considerar todas estas variables para ser capaz de maniobrar el tanque. De este modo, la inteligencia artificial tan solo tiene que regular dos variables de entrada dentro del rango [-1,1]. Siendo una de ellas la que maneja si el tanque va hacia delante o hacia atrás siendo el valor -1 máxima velocidad marcha atrás y +1 máxima velocidad hacia delante, y la otra variable con el mismo rango siendo -1 girar a toda velocidad hacia la izquierda y +1 girar a toda velocidad hacia la derecha. Estas variables simplemente indican el porcentaje de fuerza que deben hacer los motores para que el tanque respetando las leyes de la física que en el caso de la herramienta utilizada Unity3D está implementado por el motor de físicas PhysX.

Para detectar posibles obstáculos frente al agente, éste cuenta con una operacion denominada **BoxCast**[34]. Esta operación consiste en "deslizar una caja imaginaria"desde un punto en una dirección hasta una distancia deseada. De forma similar a lo que haría un sensor de ultrasonidos, los agentes de este trabajo utilizan esta operación a modo de sensor deslizando una caja del mismo volumen y forma aproximada del agente hacia delante. La caja se desliza desde la posición del agente hasta 8 metros por delante y se comprueba si esa caja ha chocado contra algo. En caso de haber chocado contra algo se toma una acción distinta en función de aquello que se ha encontrado. Si es un obstáculo pequeño simplemente se ignora. En la figura 3.2 se puede observar la caja verde que representa la posición inicial de la caja que se va a deslizar, y la caja azul que representa la posición final de la caja tras haberse deslizado. La línea blanca es el recorrido de la caja desde la posición inicial hasta la posición final. Tras haber hecho esta operación, la función nos devuelve si la caja ha chocado con algún obstáculo y además nos da información adicional sobre el choque. La información que nos devuelve la función son vectores como por ejemplo la normal del choque y la distancia que hay hasta el obstáculo. Si es un obstáculo grande se corrige la dirección del agente para que que comience a rodear por la dirección más favorable a la dirección actual del tanque sin tener en cuenta el camino más corto, ya que el agente no tiene la capacidad de obtener información sobre cual es el lado más corto para rodear.

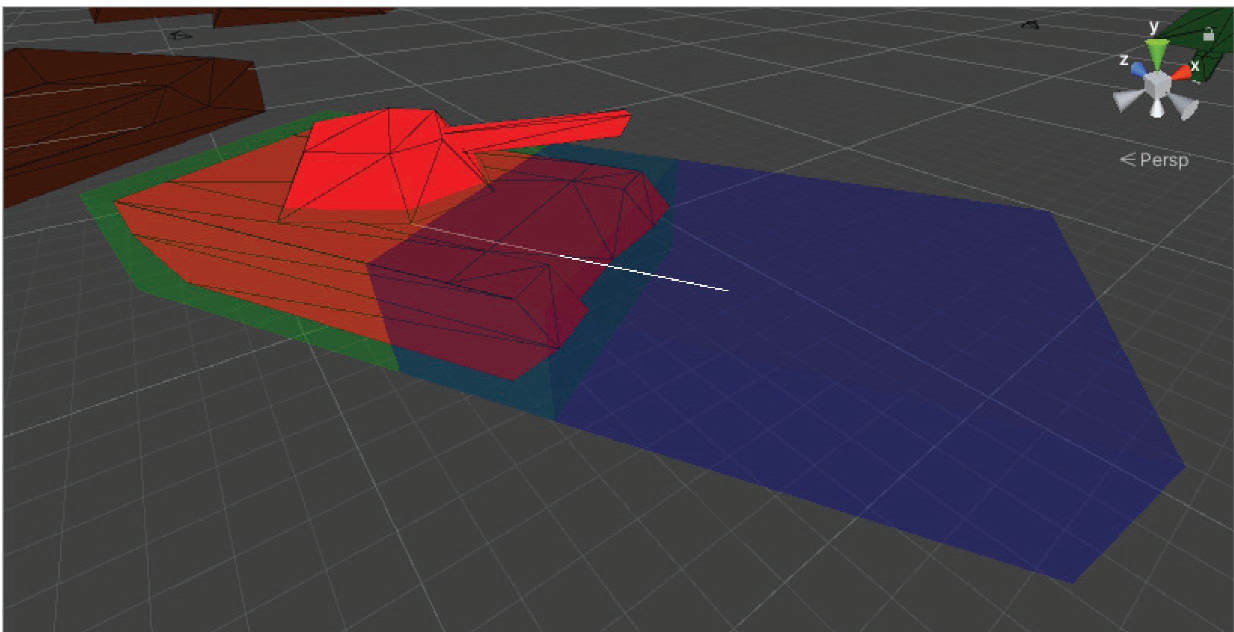

Figura 3.2: Figura que representa el BoxCast.

Con el objetivo de que el agente pueda discernir cual es la dirección más favorable a su marcha a la hora de encontrarse con un obstáculo y poder cumplir con la finalidad de rodear obstáculos además de ser capaz de mantener cierta distancia con la pared del obstáculo que está rodeando, utiliza en las cuatro diagonales la operación **RayCast**[33]. Esto nos permite ver el punto de intersección de una recta con un obstáculo. En la siguiente figura 3.3 se puede ver que hay cuatro líneas saliendo del centro del tanque hacia sus extremos. El color simplemente es para diferenciar las dos de color azul que están en la parte delantera y las magentas que están en la parte trasera. Para ahorrar cómputo, si el tanque está llendo marcha adelante solo se realizan las operaciones raycasts azules ya que si va en ese sentido los raycasts que van a detectar si se va a chocar con algo son los azules. En caso de que el tanque esté conduciendo marcha atrás, los raycasts azules se ignoran tomando como referencia los magentas que cumplen exactamente la misma función pero en el sentido contrario. Si uno de estos raycasts detecta un obstáculo, la dirección del tanque se ajusta de manera perpendicular al obstáculo y avanza hasta dejar de detectarlo. En este caso se vuelve a girar en la dirección del objetivo. En caso de que que existiera la mala suerte de que ambos raycasts detectan simultáneamente un obstáculo, se da prioridad al sensor izquierdo haciendo que el tanque rodee la pared de izquierda a derecha en sentido contrario a las agujas del reloj.

El peor caso posible sería que un tanque se encontrase a otro rodeando el mismo obstáculo en dirección contraria. En este caso ambos tanques al detectarse mutuamente se detendrían unos instantes. Para solucionar este problema cada tanque de manera independiente genera un tiempo de espera aleatorio de entre cinco y doce segundos. Tras pasar este tiempo, los tanques empezarán a girar en un sentido aleatorio hasta detectar que su parte delantera está libre. El primer tanque en encontrar una posición libre empezará a moverse en esa dirección dejando espacio provocando que el otro tanque pueda pasar. Si aun así ambos tanques decidieran moverse de manera simultanea que sería la peor suerte posibe dentro de este caso, ambos

acabarían volviendo por el camino por el que han venido. De esta forma acabarían rodeando el obstáculo posiblemente de la forma más ineficaz.

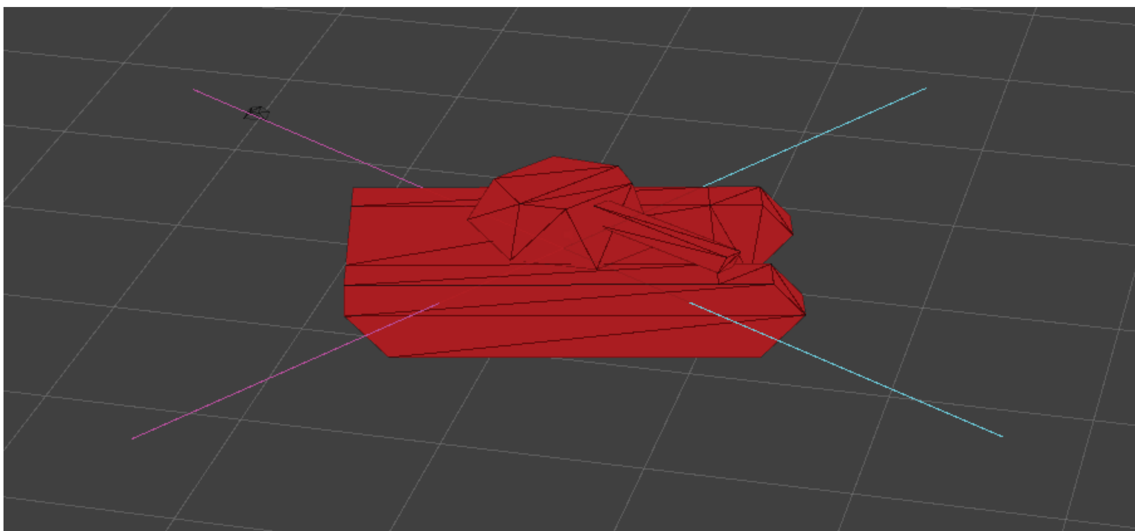

Figura 3.3: Figura que representa los RayCasts.

3.1.4. Descripción del comportamiento

Para definir el comportamiento del agente que controla los tanques distribuidos por el entorno se ha optado por una matriz de interacción que se puede ver en la figura 3.4. Esta matriz define el comportamiento del agente en función de los obstáculos, otros agentes o el mismo jugador. En este proyecto todos los elementos son considerados agentes. Estos elementos son los que el agente puede encontrarse durante la ejecución del programa.

Reglas de la matriz:

- **Tocar:** Si el agente **Árbol** es tocado de alguna forma este se tumba.
- **Apuntar:** Regla que maneja el sistema de apuntado del tanque seguido de la distancia mínima de apuntado y la prioridad.
- **Atacar:** Getiona cuando debe disparar el cañón. Seguido del ángulo de ataque y la prioridad.
- **Esperar:** Espera de un tiempo aleatorio para dar tiempo a que un tanque aliado deje libre el paso. Seguido del mínimo y el máximo tiempo aleatorio de espera y la prioridad.
- **Rodear:** Rodeo del obstáculo.

Aquellas reglas de la matriz que se encuentran en la misma matriz y tienen la misma prioridad pueden ejecutarse simultáneamente.

Destino / Origen	⊖					
	Controlado por el jugador					(Tocar;0)
	(IrAlJugador;0)	(Apuntar;240;0) (Atacar;240;0) (Emboscar;15;1)	(Esperar;2)	(Rodear;0)	(Rodear;0)	(Tocar;0)
		(Apuntar;350;0) (Atacar;350;0)				

Figura 3.4: Matriz de interacción de EXECUTANKS.

Junto con la matriz de interacción, para el desempeño de los comportamientos de los agentes se aplica el uso de varias fórmulas para controlar la aceleración del motor del tanque en función de los eventos gestionados por la dicha matriz. Para provocar pequeñas diferencias en el comportamiento de los agentes y que no parezcan copias exactas los unos de los otros se aplican ligeras variaciones de manera individual y aleatoria en cada uno de los agentes. Esto se provoca, sumando o restando pequeñas cantidades a las variables, provocando así que los agentes se comporten de manera ligeramente distinta haciendo que giren y se muevan a velocidades un poco distintas.

3.1.5. Control de velocidad

El control de velocidad del tanque está controlado por un radio de aceptación en el cual consideramos haber llegado al objetivo. Una constante que controla la pendiente de deceleración, y la distancia calculada entre la pocisión actual del angente y la posición objetivo. Hay que mencionar que la imagen de la función está saturada entre los valores 0 y 1 ambos inclusive ya que lo que representa esta función es la aceleración del motor del tanque. Siendo 0 totalmente parado y 1 máxima potencia.

- d:= Distancia a la que se encuentra el agente del objetivo.

- p:= Pendiente de deceleración.
- a:= Distancia a partir de la cual consideramos estar en el objetivo.

$$f(d, r, a) = (d - a)/p \tag{3.1}$$

Para comprender mejor el comportamiento de la función en la figura 3.5 está representado para un dominio pequeño el control de aceleración. Como podemos ver, a partir del 3 que es cuando el agente deja de estár en el objetivo, la aceleración empieza a aproximarse a 1 según la pendiente definida en la variable P. Los valores negativos no están representados por que no tiene ningún sentido medir distancias negativas. Para todos los valores hasta el infinito a partir del 5 que es donde la gráfica llega a su valor máximo se saturan el límite a 1 para que el agente no pueda ir a más velocidad de la que el motor simulado del tanque proporciona.

Valores utilizados para la gráfica:

- d:= Variable distancia.
- p:= 2
- a:= 3

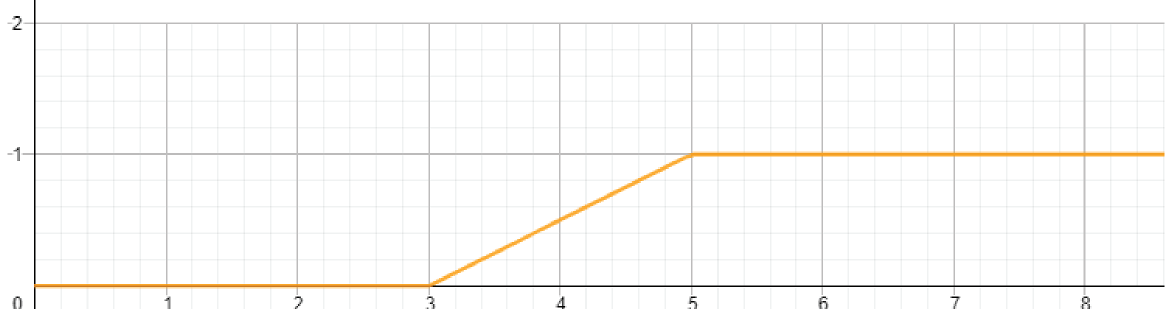

Figura 3.5: Gráfica del control de aceleración. En el eje horizontal está la distancia al objetivo y en el vertical entre 0 y 1 la potencia del motor.

Como el comportamiento de los tanques incluye que puedan maniobrar tanto hacia delante como hacia atrás, en caso de que el objetivo haya cambiado de posición y sea más fácilmente accesible marcha atrás se aplicara la misma fórmula con el signo invertido. De esta forma la imagen de la función estará definida entre -1 y 0 representando que el motor está funcionando en sentido inverso. Por último, en el caso de encontrar un obstáculo en movimiento en la dirección del agente como puede ser un tanque aliado, la aceleración del motor se fuerza a 0 para evitar una posible colisión y ceder el paso al agente. Una vez que aquello que estababa en frente de manera temporal deja de obstaculizar el camino, la aceleración del tanque vuelve a ser controlada por la formula anterior.

3.1.6. Control de giro

De manera análoga al control de velocidad,la fórmula es la misma sustituyendo la distancia por el ángulo entre la dirección del tanque y la dirección del destino requerido.Además el control de giro también se representa entre 0 y 1, representando

el 0 como que no hay que girar nada, 1 sería girar complétamente a la derecha, y -1 girar completamente hacia la izquierda. En este caso existe la excepción de que la torreta no tiene interaccion alguna hasta encontrarse lo suficientemente cerca del objetivo del agente.

- d:= Ángulo entre el objetivo y la dirección del agente
- f:= Pendiente de deceleración
- a:= Ángulo a partir del cual se considera el agente alineado en su trayectoria.

El control de giro de por si no controla ni encuentra una ruta, si no que tan solo se centra en que la trayectoria del tanque sea una linea recta entre la posición en la que se encuentra y el lugar objetivo actual que tiene. Este objetivo puede ser el jugador directamente en caso de que no haya ningun obstáculo entre medias, o en caso contrario el objetivo se sustituye por un objetivo temporal que guía al agente para rodear aquello que impide su paso.

$$f(d, r, a) = (d - a)/r \tag{3.2}$$

3.1.7. Control de la torreta

La torreta se controla de la misma forma que el control de giro con la salvedad de que la torreta está enganchada al cuerpo del agente. De esta forma si el agente gira la torreta verá su dirección afectada por la dirección del agente, y por lo tanto puede que en ciertas circunstancias el tiempo que tarda en alcanzar la posición objetivo depende de si el control de giro favorece o no a la dirección de la torreta.

La torreta, además se comporta de manera completamente independiente al movimiento del tanque. La torreta simplemente empieza a disparar cuando el objetivo está a una distancia determinada por una variable. En primera instancia para con el fin de aumentar la precisión del disparo de los agentes, la torreta era capaz de hacer una predicción del movimiento del jugador basada en la velocidad de los proyectiles y la dirección del jugador. De esta forma los agentes eran capaces de disparar hacia donde el objetivo que en este caso es el lugar hacia donde el jugador se dirigía. Esto ocasionaba que dada la torpeza de movimiento del tanque del jugador y de los tanques en general del juego fuera prácticamente imposible esquivar los disparos. Debido a esto, este mecanismo de puntería para los agentes fué descartado ya que era prácticamente imposible que el juegador pudiera sobrevivir.

3.1.8. Comportamientos inesperados manifestados

Durante el desempeño del desarrollo de la inteligencia artificial se fueron haciendo pruebas con pocos individuos para comprobar de una forma sencilla el correcto funcionamiento de los elementos individuales que componen la implementación y verificar de esta forma que al menos los agentes se comportaban de forma correcta de manera independiente.

Tras comprobar que el comportamiento independiente funcionaba de manera correcta, se analizó el comportamiento de estos agentes en grandes cantidades. Para analizar el comportamiento de estos agentes, estos fueron distribuidos de varias formas sobre el escenario. Estas distribuciones fueron las siguientes mencionadas:

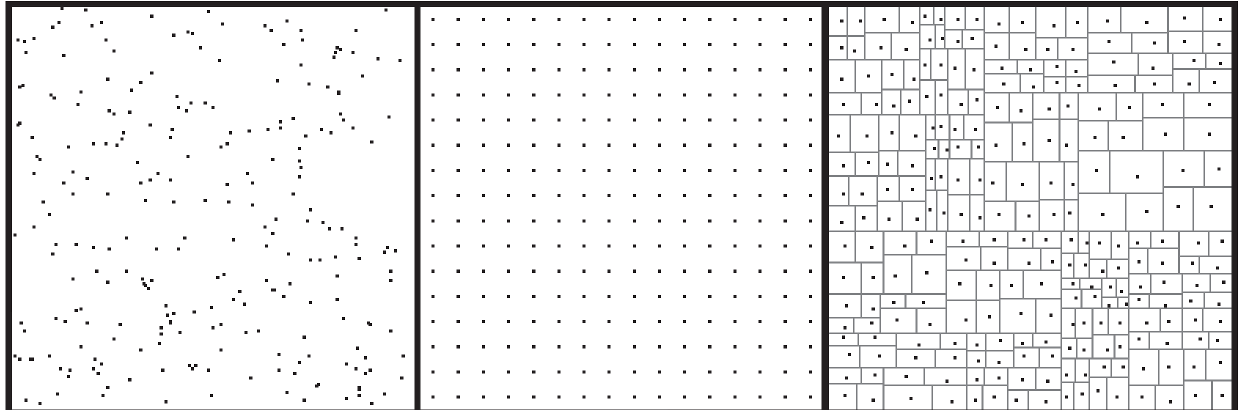

Figura 3.6: Figura que muestra las distribuciones usadas para las pruebas.

Una aleatoria, que causó pequeños problemas por que en ocasiones hacía aparecer agentes en la misma posición donde ya había otra; en forma de rejilla regular, que solucionaba el problema de la distribución aleatoria debido al patrón regular de la rejilla cuadrada; y finalmente dividiendo el espacio en forma de árbol bipartido con un tamaño aleatorio de las ramificaciones. De esta última forma los tanques aparecen colocados de forma aleatoria dentro del área de cada cuadrado sin posibilidad de aparecer dentro de otro y sin un patrón de rejilla tan reconocible.

Para todas las pruebas hechas con las distribuciones mencionadas en el párrafo anterior, y que podemos ver en la figura 3.6 y con una cantidad suficiente de individuos, aproximadamente a partir de cuatro o cinco, se empiezan a hacer apreciables los siguientes comportamientos que están analizados en la lista a continuación:

Comportamientos analizados:

- **Rodeo del objetivo:**

  Este comportamiento ocurre debido a que los agentes una vez llegan al objetivo, tienen una rutina que pasado un breve lapso de tiempo, deciden aleatoriamente si rodear el objetivo por la derecha o por la izquierda a no ser que se encuentre un agente de su mismo tipo al que ceder el paso. De esta forma, cuando hay un agente detrás de él al que está bloqueando el paso, le permite acercarse al apartarse a otro lugar al rededor del objetivo. De esta forma, si el objetivo se queda inmóvil demasiado tiempo, los agentes a base de rodear por la izquierda o por la derecha el objetivo, acabarán cediendo el paso a los agentes que están detrás y acabarán por rodearlo.

- **Comportamiento de persecución del objetivo en manada**

  El comportamiento de persecución en manada surge en parte debido al espacio que queda entre los agentes al cederse el paso entre si, y al hecho de que el objetivo es móvil y los agentes se van agrupando poco a poco en la cola de la trayectoria del objetivo. En un grupo de agentes que se han quedado acumulados por alguna razón, aquellos que no estén en el exterior del grupo, probablemente tengan una mobilidad reducida al estar rodeado por agentes de su mismo tipo al que deben de ceder el paso. Aquellos agentes que se encuentren en el exterior del grupo podrán maniobrar libremente hasta poder encontrar una zona sin agentes que obstaculicen su paso. Al ocurrir esto, de manera similar a

lo que ocurre en un atasco de tráfico, estos agentes empezarán a alejarse del grupo permitiendo que los agentes que se encuentran tras ellos vuelvan a tener suficiente espacio para maniobrar provocando que estos también avancen en la dirección del objetivo.

- **Esquive de aliados**

  El esquive de aliados aunque inicialmente no fué programado de manera intencionada, es un comportamiento que sucede debido a la inercia de los agentes. A pesar de que un agente al encontrarse de frente contra otro agente tiene que detener el motor y esperar a que se vaya y cederle el paso, es posible que si la aceleración es máxima y se ha interpuesto otro a gente en su camino no le de tiempo a frenar. En este caso, el agente que se interpone en el camino del otro es considerado de la misma forma que un obstáculo rígido. Así que aunque no sea un obstáculo rígido, mientras se está frenando, se calcula una trayectoria como si se intentase rodear el obstáculo. De este modo, podemos apreciar que cuando dos agentes aparentemente van a colisionar o pueden colisionar, efectuan una maniobra de evasión que se asemeja al esquive de dos vehículos que van a colisionar entre sí.

- **Capacidad para ignorar obstáculos**

  Este comportamiento responde perfectamente a uno de los objetivos fundamentales del desarrollo de Executanks. Este objetivo era la necesidad de una inteligencia artificial capaz de soportar un funcionamiento correcto frente a grandes cantidades de pequeños obstáculos en el escenario, que se generan en tiempo real durante la ejecución del videojuego.

  El comportamiento de estos agentes está simulado por el motor de físicas de Unity3D PhysX [32]. Esto hace que el movimiento los agentes respete las leyes de la física. El controlador que gestiona el movimiento del agente se basa en las fuerzas aplicadas por el motor simulado del tanque y no en normas arbitrarias totalmente rígidas. Esto permite que el agente tras reaccionar a posibles golpes que desplacen su trayectoria, pueda recuperar su comportamiento con normalidad, haciendo las maniobras necesarias. De esta forma, el agente al encontrarse con pequeños obstáculos que no suponen un cambio necesario en la trayectoria, símplemente puede ignorarlos pasándolos por encíma o arrastrandolos.

Capítulo 4

Resultados y conclusiones

En este apartado se comentan los puntos fuertes y carencias que podemos encontrar en este diseño, las razones que causan estas buenas o malas características y las posibles lineas futuras de investigación. Además se sugieren algunas posibiles soluciones para aquellas características de peor calidad.

- **Conclusiones sobre el rendimiento:**

  Debido al bajo coste de las operaciones realizadas es posible tener una gran cantidad de agentes sin demasiado coste de cpu. Los agentes tampoco requieren ningún tipo de preprocesamiento del entorno como si lo requieren los pathfindings basados en A* o HPA*. Estos útimos si que necesitan una rejilla o un navmesh o algún tipo de representación basada en nodos que es necesario generar para cada entorno.

- **Conclusiones sobre procesamiento de osbtáculos:**

  Los obstáculos no suponen un gran coste adicional ya que la manera principal de encontrar la ruta hacia el objetivo consiste simplemente en rodear aquello que se interpone en el camino del agente. Aunque esta premisa cumple correctamente el propósito deseado para el videojuego diseñado, que es soportar una gran cantidad de obstáculos sin añadir un gran coste de cpu por el procesamiento de los agentes, no será de gran efectividad para laberintos complicados u obstáculos cóncavos.

- **Conclusiones sobre el movimiento en grupo:**

  El método que implementa este agente en particular no es para nada efectivo en este ámbito ya que simplemente se basa en detenerse si se encuentra a un agente interrumpiendo su camino. Éste método no es de gran eficacia a la hora de compartir una ruta con muchos agentes ya que se entorpecen el paso entre ellos. Una posible solución a este problema podría ser cambiar este tipo de comportamiento por uno que haga al agente acompañar al que está bloqueando su paso. Otra posibilidad sería combinar el enrutador del agente con algún tipo de algoritmo de flocking para intentar mejorar este comportamiento.

- **Conclusiones sobre el pathfinding:**

  El sistema empleado no hace ninguna labor en especial para encontrar rutas óptimas. En este aspecto hay bastante trabajo por desarrollar. Los agentes al

limitarse a rodear los obstáculos con los que se encuentran es posible que tomen rutas que no son para nada óptimas. Aunque no es un problema grave para los casos requeridos de la aplicación para la cual se diseñó esta inteligencia artificial, no será efectivo para obstáculos cóncavos ni laberintos. Ya que esto podría llegar a hacer que los agentes queden bloqueados si se encuentran con otro que impida su paso. Quizas el empleo de algoritmos como 'E-Bug'[13] o 'el de las hormigas' mejore significativamente este problema sin afectar de manera negativa el coste del procesamiento de obstáculos.

Bibliografía

[1] Carole Bernon,Marie-Pierre Gleizes,Sylvain Peyruqueou,Gauthier Picard
ADELFE: A Methodology for Adaptive Multi-agent Systems

[2] Zahia Guessoum
Adaptive agents and multiagent systems

[3] Anais Garrell, Oscar Sandoval-Torres and Alberto Sanfeliu
Adaptive Multi Agent System for Guiding Groups of People in Urban Areas

[4] Marco Dorigo,Mauro Birattari,Thomas Stützle
Ant Colony Optimization

[5] George Kelly, Hugh McCabe
A Survey of Procedural Techniques for City Generation

[6] Ross Graham, Hugh McCabe, Stephen Sheridan
Pathfinding in Computer Games

[7] Michael Wooldridge
An Introduction To Multi Agent Systems

[8] Karl Tuyls,Ann Nowe,Zahia Guessoum
Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning

[9] Intelligent Agents: Multi-Agent Systems
Alfredo Garro,Max Mühlhäuser,Andrea Tundis

[10] Wai Kin Victor W. K. Chan
Foundations of Simulation Modeling

[11] A multiagent system based on Unity 4 for virtual perception andwayfinding
Christian Becker-Asano, Felix Ruzzoli, Christoph Hölscher, Bernhard Nebel

[12] Georgios N. Yannakakis and Julian Togelius Artificial Intelligence and Games

[13] Linda Dib.
E-Bug: New Bug Path-planning algorithm for autonomous robot in unknown environment.

[14] Michele Colledanchise and Petter Ogren.
Behavior Trees in Robotics and AI An Introduction.

[15] Ramiro A. Agis, Sebastian Gottifredi, Alejandro J. García.
An event-driven behavior trees extension to facilitate non-player Multi-Agent Coordination In Video Games.

[16] Jeff Orkin.
Applying Goal-Oriented Action Planning to Games

[17] Jeff Orkin.
Three States and a Plan: The A.I. of F.E.A.R.

[18] Nuria Pelechano, Carlos Fuentes.
Hierarchical Path-Finding for Navigation Meshes (HNA*)

[19] Xiao Cui,Hao Shi.
A*-based Pathfinding in Modern Computer Games

[20] Adi Botea,Martin Müller.
Near optimal hierarchical path-finding (HPA*)

[21] Ryan Marcotte,Howard J. Hamilton.
Behavior Trees for Modelling Artificial Intelligence in Games: A Tutorial

[22] James A. Douthwaite,Shiyu Zhao,Westlake University,Lyudmila Mihaylova Velocity Obstacle Approaches for Multi-Agent Collision Avoidance

[23] Stepan Dergachev,Konstantin Yakovlev,Ryhor Prakapovich3 A Combination of Theta*, ORCA and Push and Rotate for Multi-agent Navigation

[24] Yoann Kubera, Philippe Mathieu, Sébastien Picaul IODA: An interaction-oriented approach for multi-agentbased simulations

[25] Philippe Mathieu, David Panzoli ,S'ebastien Picault Virtual Customers in a Multiagent Training Application

[26] Philippe Mathieu, Sébastien Picault The Galaxian Project : A 3D Interaction-Based Animation Engine

Documentación de Unity:

[27] https://docs.unity3d.com/Manual/nav-AreasAndCosts.html

[28] https://docs.unity3d.com/ScriptReference/AI.NavMeshAgent.html

[29] https://docs.unity3d.com/es/530/Manual/class-NavMeshAgent.html

[30] https://docs.unity3d.com/Manual/class-NavMeshSurface.html

[31] https://docs.unity3d.com/Manual/class-OffMeshLink.html

[32] https://docs.unity3d.com/Manual/UpgradeGuide5-Physics.html

[33] https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

[34] https://docs.unity3d.com/ScriptReference/Physics.BoxCast.html

Documentación de UE4:

[35] https://docs.unrealengine.com/en-US/API/Runtime/
AIModule/FGraphAStar/index.html

[36] https://docs.unrealengine.com/en-US/Basics/Components/Navigation/index.html

[37] https://docs.unrealengine.com/en-US/Resources/ContentExamples/NavMesh/1_2/index.html

[38] https://docs.unrealengine.com/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreesOverview/index.html

Documentación de Valve:

[39] https://developer.valvesoftware.com/wiki/Navigation_Meshes#Area_types

Anexo

Este capítulo es opcional, y se escribirá de acuerdo con las indicaciones del Tutor.