



# IA de videojuegos

27/06/2021

Javier Sánchez Melgarejo  
Javier Nicolás Hernández  
Eduardo González Sevilla  
Grupo 1 GLaDOS

## 1. ÍNDICE

<b>ÍNDICE</b>	<b>1</b>
<b>INTRODUCCIÓN</b>	<b>2</b>
<b>SOFTWARE Y HARDWARE UTILIZADO</b>	<b>4</b>
<b>EXPLICACIÓN SOBRE ELEMENTOS DESARROLLADOS</b>	<b>5</b>
3.1. BLOQUE 1: MOVIMIENTO	5
Steering	7
Body	9
Agent	9
AgentPlayer	10
AgentNPC	10
Elementos implementados	13
3.2. BLOQUE 2: IA Estratégica y táctica	34
Sistema de Combate	34
Escenario	34
Condiciones de Victoria	35
Mapa Táctico	35
Comportamiento de los agentes	37
Guerra Total	39
Pathfinding Táctico	40
<b>INTERFAZ DE USUARIO</b>	<b>40</b>
Modos de comportamiento	40
Pantalla de victoria	41
<b>DIAGRAMAS QUE REFLEJEN LA IA TÁCTICA IMPLEMENTADA</b>	<b>42</b>
<b>GUÍA DE USO DEL PROYECTO</b>	<b>44</b>
Formación	45
<b>CONCLUSIONES</b>	<b>45</b>
<b>REFERENCIAS</b>	<b>49</b>

## 1. INTRODUCCIÓN

En este documento se recopila la documentación así como el manual de usuario del trabajo de prácticas de la asignatura IA para el desarrollo de videojuegos de la Facultad de Informática de Murcia en el curso 2020/2021 para la convocatoria de Julio.

La práctica consiste en implementar algunos elementos de inteligencia artificial. En un primer bloque se desarrollarán los elementos asociados a movimiento. En otro bloque se mostrarán elementos de IA táctica y estratégica en un entorno de juego de guerra en tiempo real.

En la primera parte del proyecto implementamos diferentes comportamientos de los personajes, como puede ser que un personaje siga a otro o que un personaje evite chocar con una pared. El desarrollo de esta fase nos ayuda a comprender mejor qué hay detrás de un videojuego y cómo podemos simular inteligencia en los personajes del mismo.

En una segunda parte creamos una escena más general donde mezclamos varios comportamientos desarrollados en la fase anterior y conseguimos permitir a un usuario enfrentarse a la máquina. El escenario consiste en un terreno con dos equipos, cada uno con su base y sus unidades de ataque. El objetivo es destruir la base enemiga.

Como resultado de este enfrentamiento veremos como la máquina es capaz de simular inteligencia con acciones de defensa o de ataque y con la funcionalidad implementada en la fase primera del proyecto.

La estructura del documento se divide en seis partes. En la siguiente sección empezaremos encontrando el software y hardware utilizado por el grupo durante el desarrollo.

La tercera sección es la más importante del documento ya que contiene las decisiones de diseño y explicaciones sobre los elementos implementados en nuestro proyecto. Mostrarímos diagramas de clase, imágenes de ejemplo y algunos problemas encontrados.



En las secciones finales del documento detallaremos la interfaz de usuario usada y un manual de usuario para indicar el uso de algunos elementos del proyecto.

El trabajo ha sido realizado por los alumnos: Javier Sánchez Melgarejo, Eduardo González Sevilla y Javier Nicolás Hernández.

## 2. SOFTWARE Y HARDWARE UTILIZADO

Para la realización de la práctica decidimos usar Unity como motor de juegos. El motivo principal es el ahorro en tiempo que supondría aprender un nuevo motor de juegos, ya que aunque no somos expertos en Unity lo hemos visto un poco en la asignatura *Fundamentos Computacionales de los Videojuegos* del primer cuatrimestre de este mismo curso.

La versión utilizada en el desarrollo es 2020.3.12f1 al ser la versión con las últimas funcionalidades y con *Long-Term-Support*, en el momento del desarrollo del proyecto.

Respecto a la versión de sistema operativo usamos Windows 10 al considerar que ofrece mayor comodidad con Unity y al ser el sistema operativo más utilizado en el mundo del desarrollo de videojuegos.

Por último, en las especificaciones hardware usadas por los miembros del equipo tenemos procesadores i5 e i7, capacidades de 8 y 16 Gb de memoria RAM y gráficas Nvidia 960M o superiores.

Esta sección es únicamente informativa ya que los requisitos para usar Unity son menores.

### 3. EXPLICACIÓN SOBRE ELEMENTOS DESARROLLADOS

En esta sección explicaremos cómo hemos abordado los elementos obligatorios y opcionales que podemos encontrar en nuestro proyecto y cuáles han sido las decisiones de diseño tomadas en su desarrollo.

#### 3.1. BLOQUE 1: MOVIMIENTO

Comenzaremos mostrando nuestra jerarquía de clases y comentando los elementos más importantes de cada una.

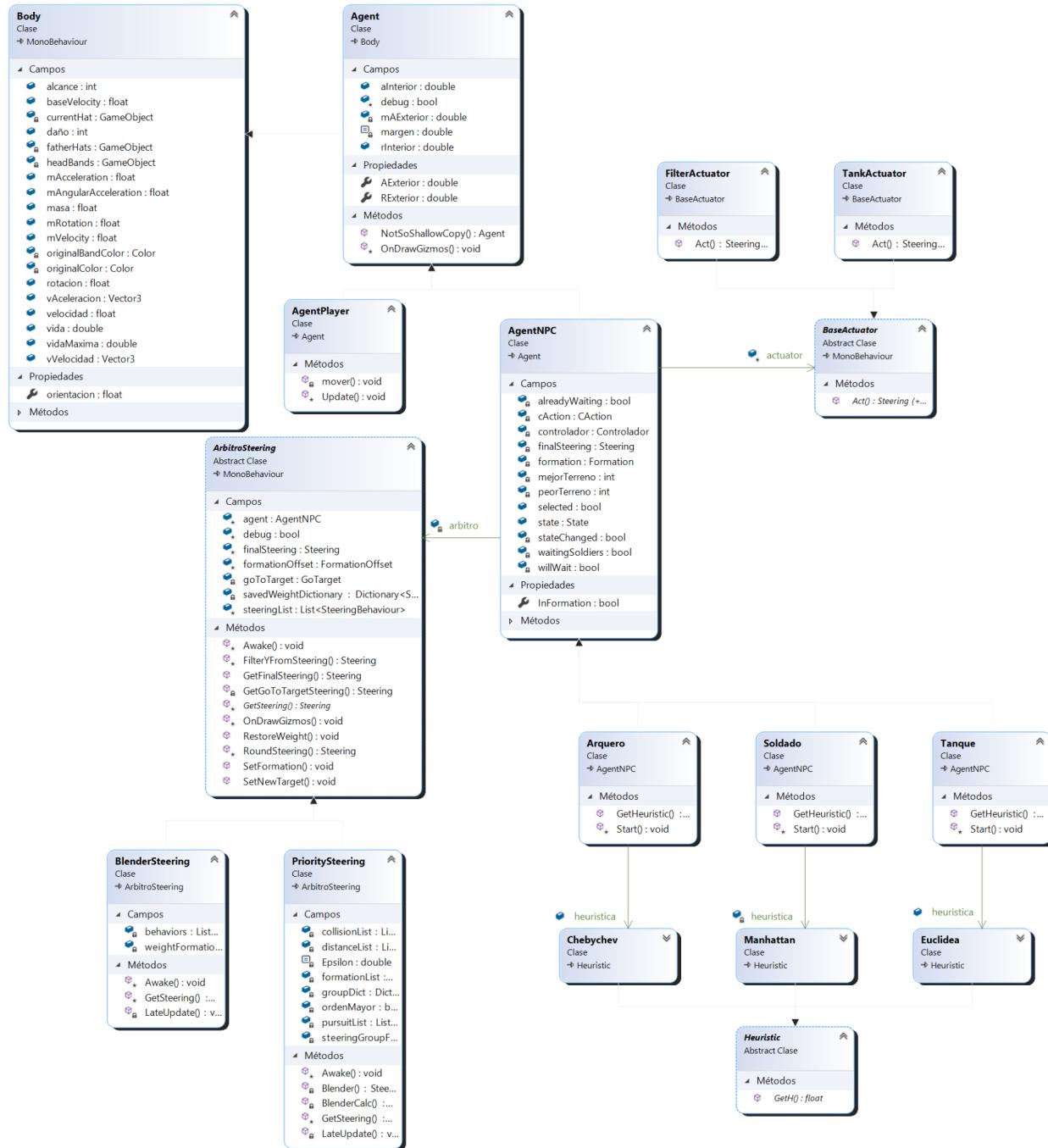


Figura 1. Únicamente mostramos los atributos de cada clase.

La clase principal es **Body**, de la cual heredará **Agent**. A su vez tenemos dos tipos de agentes, **AgentNPC** y **AgentPlayer**. Esta última será padre de nuestros tres tipos de unidades; arquero, soldado y tanque. Además nuestro agenteNPC usará árbitros. Tenemos dos tipos de árbitros en

nuestro diseño: *BlenderSteering* y *PrioritySteering*. La clase de la que heredan ambos es *ArbitroSteering*.

También hemos implementado el uso de dos actuadores por *AgentNPC*: *FilterActuator* y *TankActuator*, cuya clase padre es *BaseActuator*.

## Steering

Un sistema steering (comportamiento sobre la dirección) es un sistema que propone movimientos a los agentes en base a sus entornos locales, es decir, utilizando la información sobre el mundo en función de lo que perciben sus sentidos. Un sistema steering controla y aplica el movimiento del agente, prestando atención a los objetivos múltiples como no tropezar con paredes y otros agentes, al tiempo que se persigue o evitan agentes específicos, o tratando de mantenerse dentro de un grupo.

Para implementar un steering usamos una estructura. El componente *lineal* es un vector lineal que indica la velocidad o aceleración en el movimiento, y el componente *angular* es la velocidad o aceleración angular y lo usamos en las rotaciones.

En nuestro proyecto encontramos steerings básicos (que pueden actuar por sí solos), delegados (que heredan de los básicos), de grupo, de colisiones y de formación.

No todos los steerings de grupo heredan de los básicos, en nuestro proyecto *Cohesion* hereda mientras *Alignment* y *Separation* no. Los steerings de este tipo necesitan varios agentes para actuar, encontramos un ejemplo con la escena *Flock*. La forma de implementar estos steerings es a través de una lista de targets que tienen los agentes y de la que obtienen información de su alrededor y actúan en consecuencia.

Los steerings de colisiones permiten a los agentes esquivar paredes mediante el uso de bigotes, como es el caso de *Wall Avoidance*, mientras que otros usan predicciones para evitar colisionar, como es el caso de *Collision Avoidance*.

Los steerings de formaciones permiten a los soldados mantenerse en formación y moverse respecto a un líder.

En la Figura 2 podemos ver los steerings implementados en el proyecto y las relaciones entre ellos. En secciones siguientes veremos cada uno en profundidad.

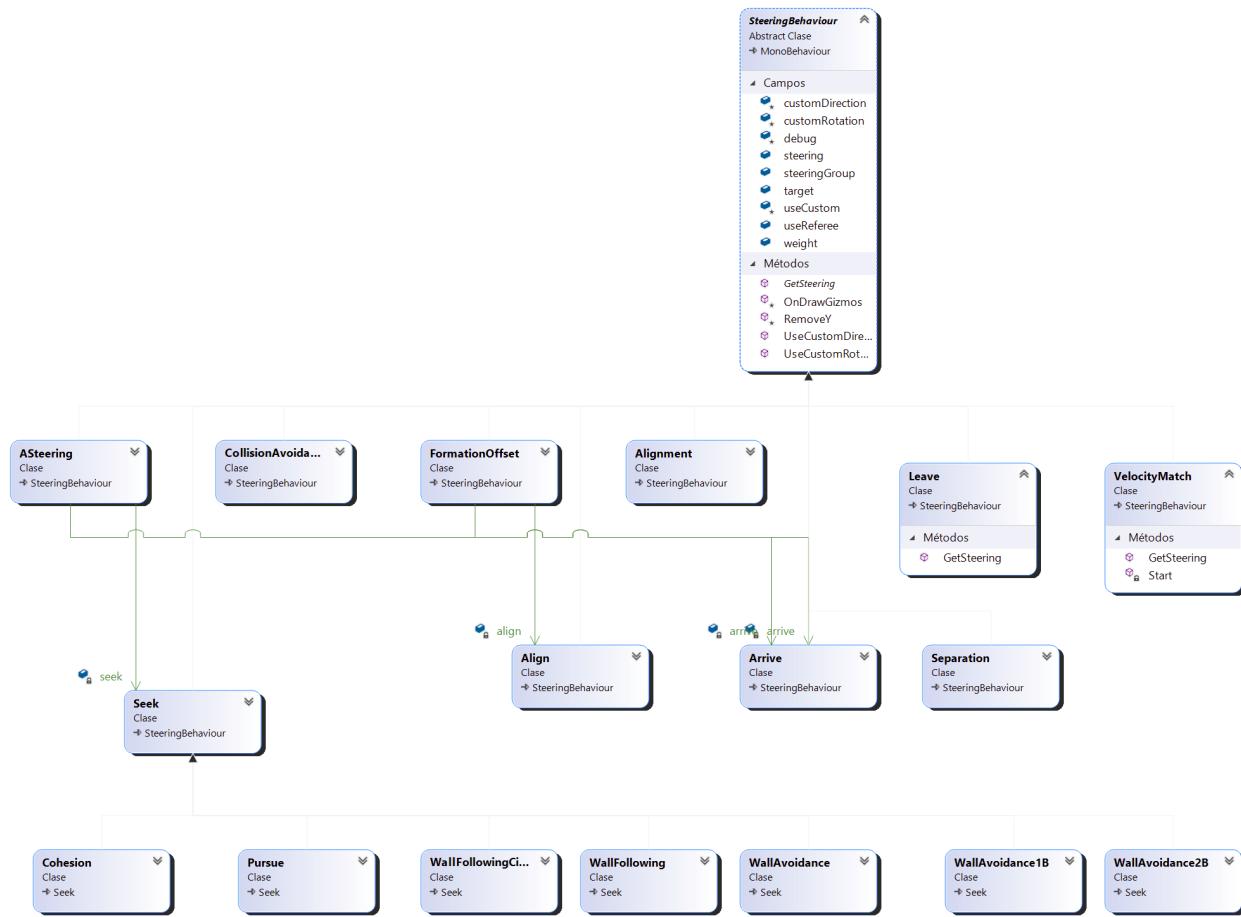


Figura 2. Diagrama de clases con los steerings.



## Body

La clase *Body* es la componente que tiene las propiedades usadas para físicas. Tiene métodos que serán usados por los agentes NPC en los steerings. Tiene varias funciones “ayudantes” que permiten calcular ángulos, vectores, posiciones y rotaciones.

## Agent

Como la posición de nuestro personaje es únicamente un punto pero nuestro agente tiene volumen necesitamos alguna manera de saber el tamaño real, en especial para poder controlar las colisiones o saber cuando llegamos a un destino.

Por este motivo en el agente añadimos dos radios, uno interior y de menor tamaño que envuelve a nuestro personaje y nos permite saber si hemos colisionado cuando un objeto entra en su perímetro, y otro exterior que nos avisa con antelación de la colisión para poder ir reduciendo la velocidad.

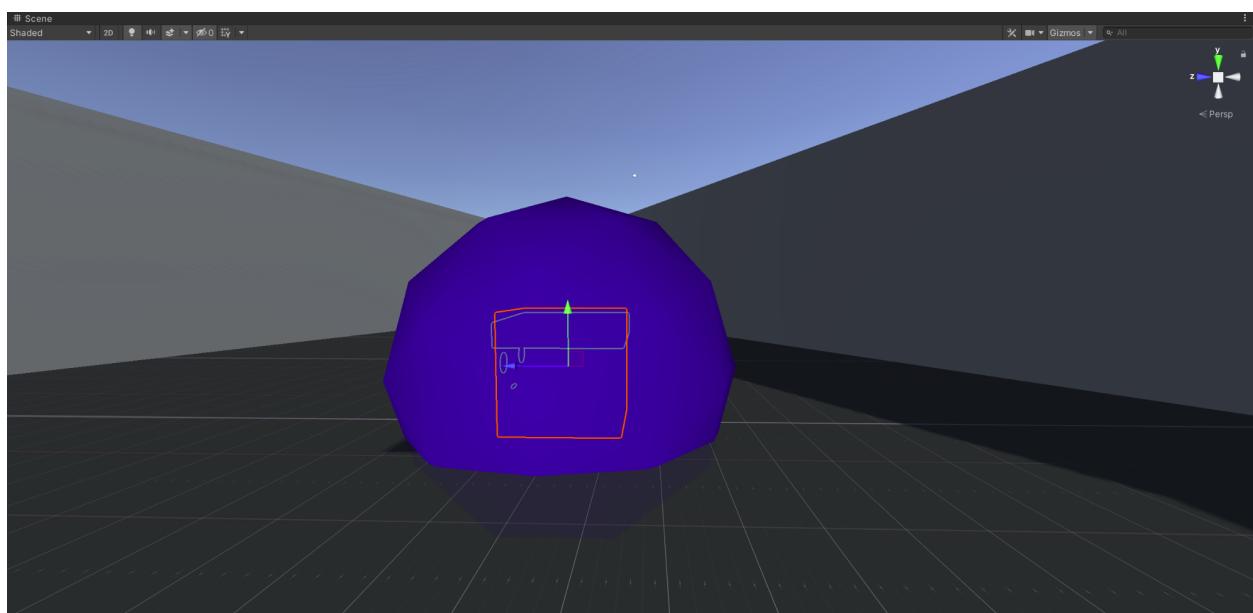


Figura 3. Mostramos radio exterior.

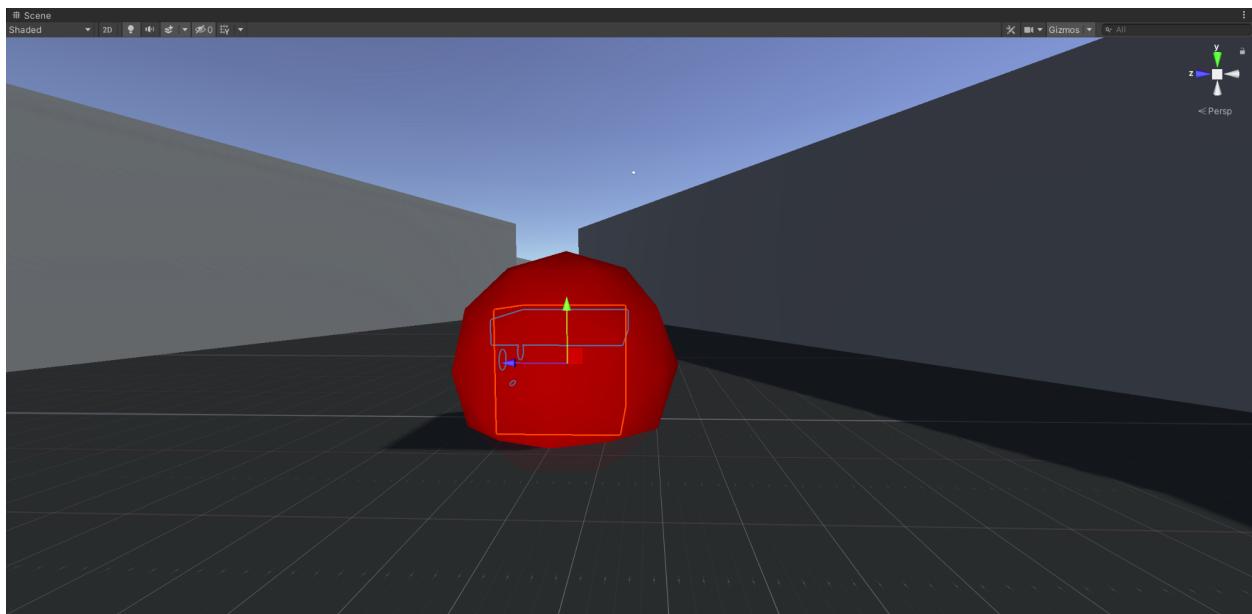


Figura 4. Mostramos radio interior.

Un ejemplo de este comportamiento lo encontramos en el steering *Arrive* para saber cuando hemos llegado al objetivo.

### **AgentPlayer**

El primer tipo de agente que encontraremos en el proyecto es *AgentPlayer*. Es el tipo más sencillo ya que únicamente se mueve con las teclas del teclado y no tiene steerings asociados.

### **AgentNPC**

Es la componente reactiva que se encarga de calcular los steerings, arbitrarlos y aplicar un actuador.

El personaje puede estar en tres diferentes estados.

- Normal: Realiza los steering que tiene asignados.
- Waiting: Ignora los steerings y se queda quieto.
- Action: Realiza una acción. La acción puede ser *Forming*, donde ignora todos los steerings excepto los de formación, *goToTarget*, donde ignora todos los steerings salvo los de ir al objetivo y *None*, que es la acción por defecto. La acción *None* va con el estado *Normal* o *Waiting*.

Las variables *state* o *cAction* son las usadas para el control de los estados.

También encontramos un flag *selected* que nos permite saber si está seleccionado para cambiar el color del personaje o hacer acciones en grupo.

*Formation* es una estructura/clase que lleva funciones para trabajar con la formación y el estado actual.

Cuando seleccionamos varios personajes se crea el objeto *formation*. Este objeto en su interior indica quien es el líder y quiénes los soldados, así como la posición relativa del líder respecto a los soldados. Lleva también funciones para saber el estado actual de la formación y pasos de mensaje entre soldados y líder y entre el controlador y la formación, esto permite que los soldados sepan si la formación se ha disuelto.

No hace la formación en sí pero permite a los steerings encargados de ello (*formationOffset*) hacerla.

El controlador es la clase que controla los personajes que están seleccionados. Le indica si tienen que ir a algún lugar o si tienen que hacer una formación. Lee del teclado y dice a los agentes lo que tienen que hacer.

La última propiedad es el *finalSteering*. Es el steering que usará el agenteNPC. Para entenderlo mejor tenemos la Figura 5.

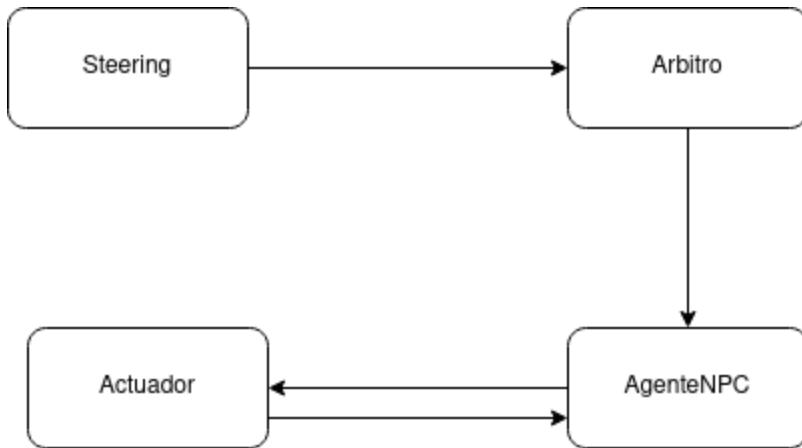


Figura 5. Interacción entre steering, árbitro, agenteNPC y un actuador.

Cuando un agenteNPC está en el estado normal, le pregunta al árbitro los steerings. El árbitro recorre la lista de steerings y dependiendo del tipo de árbitro modifica esta información aplicando pesos (*blendersteering*) o prioridades (*prioritySteering*) para obtener un steering final. El agenteNPC recibe la información del árbitro y se la manda al actuador que le devolverá el steering final filtrado que usa el agente.

Encontramos dos actuadores, uno que filtra la componente Y porque nuestros personajes no saltan, y otro que filtra el vector movimiento para que nuestro personaje se mueva únicamente hacia adelante o atrás, como un tanque.

Lo que devuelve el actuador lo utiliza el agenteNPC usando las ecuaciones del movimiento no acelerado, donde va a velocidad máxima si tiene un objetivo, o acelerado, donde empieza con velocidad cero y aumenta a la máxima.

En nuestro proyecto encontraremos tres tipos de unidades NPC: arquero, soldado y tanque. Cada uno se diferencia en la vida, velocidad y alcance. En la Figura 6 vemos los modelos elegidos para cada agente.

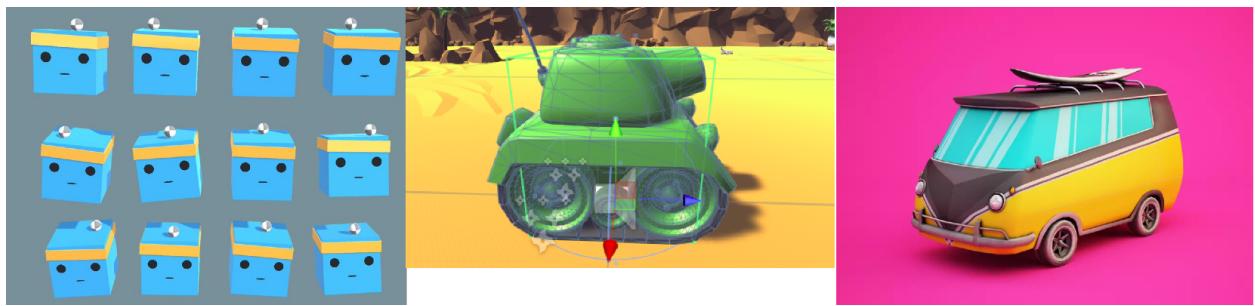


Figura 6. Los modelos elegidos para las unidades: soldado, tanque y arquero.

El último detalle del diagrama lo vemos en que cada NPC tiene una heurística diferente para ver distintos comportamientos, e.g. con el algoritmo LRTA usado en el *pathFinding*.

## Elementos implementados

Comenzamos detallando los elementos obligatorios.

- a) **Los distintos tipos de unidades deberán tener distintas velocidades de movimiento: tanto la propia de la unidad (p.e. una infantería ligera va más rápida que una pesada) y también en función del terreno, es decir, más lento cuando atraviesen terrenos más difíciles para ellas.**

Como vemos en el diagrama los tipos de unidades arquero, soldado y tanque heredan de AgenteNPC. En el agenteNPC definimos una propiedad para poder indicar la velocidad en el mismo inspector de Unity.

Tenemos tres tipos de velocidad.

- Velocidad máxima: Indica la velocidad máxima actual que puede alcanzar el personaje.
- Velocidad base: Indica la velocidad máxima inicial del personaje, con la que comienza.
- El vector velocidad, que representa el movimiento.

La velocidad base se inicializa a la velocidad máxima. En función del terreno aumenta o disminuye la velocidad máxima pero la base se mantiene.

La clase *Terrain* de Unity tiene una lista de *TerrainLayer*. Esto define la superficie del terreno. Cada *TerrainLayer* va asociado con un entero que coincide con la posición en la lista de terrenos en *Terrain*, por ejemplo, el terreno de montaña tiene el índice 1.

Las etiquetas de *mejorTerreno* y *peorTerreno* son dos enteros que indican el índice del mejor y peor terreno. Cada agenteNPC actualizará la velocidad en el método *lateUpdate()* en función del terreno donde se encuentra, de forma que:

- En mejor terreno, *velocidad máxima* = *velocidad base* \* 1.5
- En peor terreno, *velocidad máxima* =  $\frac{\text{velocidad base}}{1.5}$

Si no hay un objeto de la clase *Terrain* asociado no se actualiza la velocidad y se utiliza la velocidad máxima asignada inicialmente.

- b) **Seleccionar uno o varios personajes con el puntero/ratón.** Se puede compaginar ratón+teclado. Los personajes seleccionados se mantendrán en este estado hasta que diga lo contrario el usuario. El usuario podrá deseleccionar a todos los seleccionados usando un atajo de teclado o pulsando algún icono en la pantalla. Opcionalmente, si quiere, puede tener un proceso de deselección individual.

Para seleccionar un personaje solo hay que hacer click izquierdo sobre el, al hacerlo cambiara de color para representar que ha sido seleccionado y se podra realizar acciones sobre el. Al seleccionar se activa un flag en la clase Agent NPC indicando su estado y se introduce en una lista en el controlador.

Posteriormente si se quiere mover el personaje a un lugar se pulsa la G y se selecciona el punto del mapa con el click derecho, Esto hace que el steering *GoToTarget* se actualice con la nueva posición y que el estado del NPC pase a Acción y el tipo de acción a *GoToTarget* esto hace que el árbitro y el Npc ignoren todos los steering menos el del *GoToTarget*.

Finalmente si se quiere seleccionar el NPC basta con hacerle click o pulsar la tecla C o R, si se pulsa la C se deseleccionan todos los NPC pero siguen con su acción y estado , si se pulsa la R se resetea al personaje.

- 
- c) Implementación de los siguientes steerings básicos acelerados: Seek, Flee, Arrive, Leave, Align, anti-Align, Velocity Matching.

### Seek

Este steering se basa en perseguir al objetivo pasado por parámetro utilizando como vector velocidad la diferencia entre la posición del objetivo y del npc que implementa el steering multiplicado por la máxima aceleración del mismo. Se trata de un steering que no tiene en cuenta si llega al objetivo, por lo que acaba chocando contra el propio objetivo en la mayoría de casos.

### Flee

Es lo contrario al *Seek*. En vez de calcular el vector velocidad como la diferencia entre el objetivo y su posición actual, la calcula como la posición actual menos el objetivo, con el fin de dar un vector velocidad que vaya en dirección contraria a la del objetivo del cual huir.

### Arrive

Es una variante del *Seek*. Mientras que en el primer steering el personaje no se detiene cuando llega al objetivo, haciendo que choque con este, con el steering *Arrive* comprobamos que no estamos a una distancia marcada por el parámetro *slowRadius*. Esta variable marca el rango de un círculo alrededor del objetivo con el objetivo de, si el npc se encuentra a una distancia del objetivo menor a *slowRadius*, significa que se encuentra cerca del objetivo y debemos disminuir la velocidad, haciendo que el personaje se pare cuando esté lo suficientemente cerca del objetivo.

### Leave

Se trata de lo opuesto a *Arrive*. Al igual que con *Flee*, se huye del objetivo calculando el vector velocidad como la posición actual menos el objetivo. Como ocurre en *Arrive*, si el radio del círculo que rodea al objetivo es menor que la distancia que tenemos con el objetivo, vamos disminuyendo la velocidad. Esto hace que si el objetivo se aleja pasamos de cero a máxima velocidad, haciendo que no sea muy creíble, por lo que se suele usar *Flee* como el contrario a *Arrive*.

Podemos encontrar un ejemplo de estos cuatro steerings en la escena *Seek-Flee-Arrive-Leave*. En la Figura 7 vemos como los agentes que implementan *Seek* y *Arrive* se encuentran cerca del target, mientras que los agentes con *Flee* y *Leave* se alejan.

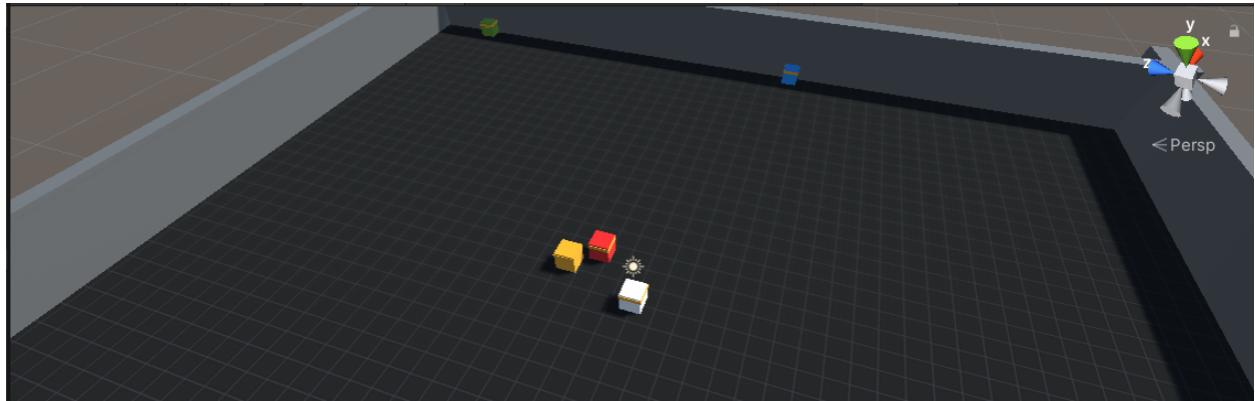


Figura 7. El target donde se dirigen los agentes es el blanco. El agente rojo implementa seek, el amarillo arrive, el azul flee y el verde leave.

### Aling

Este steering se basa en la modificación del ángulo, haciendo que el NPC mire en la misma dirección que el objetivo. Para ello calculamos la rotación como la diferencia de la rotación del objetivo con el personaje, mapeando el resultado en el rango de  $[-\pi, \pi]$ .

### Anti-Aling

Consiste en mirar en la dirección opuesta a la del objetivo. Para ello sumamos  $\pi$  a la orientación y realizamos un Aling al objetivo. En la Figura 8 vemos un ejemplo de estos steerings.

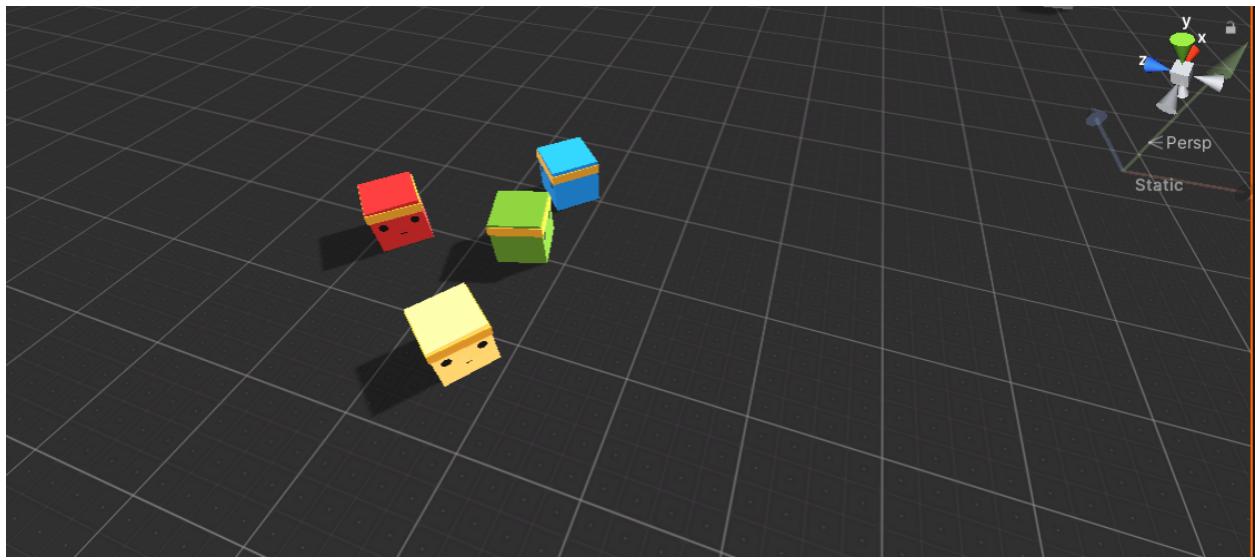


Figura 8. El agente rojo es el target. El azul implementa Face, el verde anti-aling y el amarillo aling.

### Velocity Matching

Consiste en obtener la misma velocidad que nuestro objetivo.

#### d) Implementación de los steerings delegados: Persue, Face, Wander, Path Following sin pathOffset.

##### Persue:

Se trata de un steering delegado que utiliza el *Seek*. En este caso antes de perseguir al objetivo mediante *Seek*, primero se calcula la posición en la que estará dicho objetivo en el futuro teniendo en cuenta la velocidad actual del objetivo con el fin de ir a esa posición en lugar de perseguir al objetivo, dando una sensación de inteligencia por parte del NPC al querer atajar camino. Para ello calculamos nuestro índice de velocidad predecida, usando nuestra velocidad actual y la distancia con el objetivo. Una vez tenemos dicho índice, calculamos la nueva posición del objetivo como su posición actual sumado a su velocidad por el índice de predicción. Finalmente, se hace un *Seek* a esa nueva posición calculada.

##### Evade:

Lo opuesto a *Persue*. Trata de predecir de la misma manera dónde se encontrará el objetivo para hacer un *Flee* con la posición predecida del objetivo.

Face:

El agente calcula la diferencia entre el vector que mira y el de la posición angular a futuro de su objetivo. De esta forma el agente mirará a su target.

Encontramos un ejemplo de los steerings Persue, Evade, Face y Wander en Figura 9.

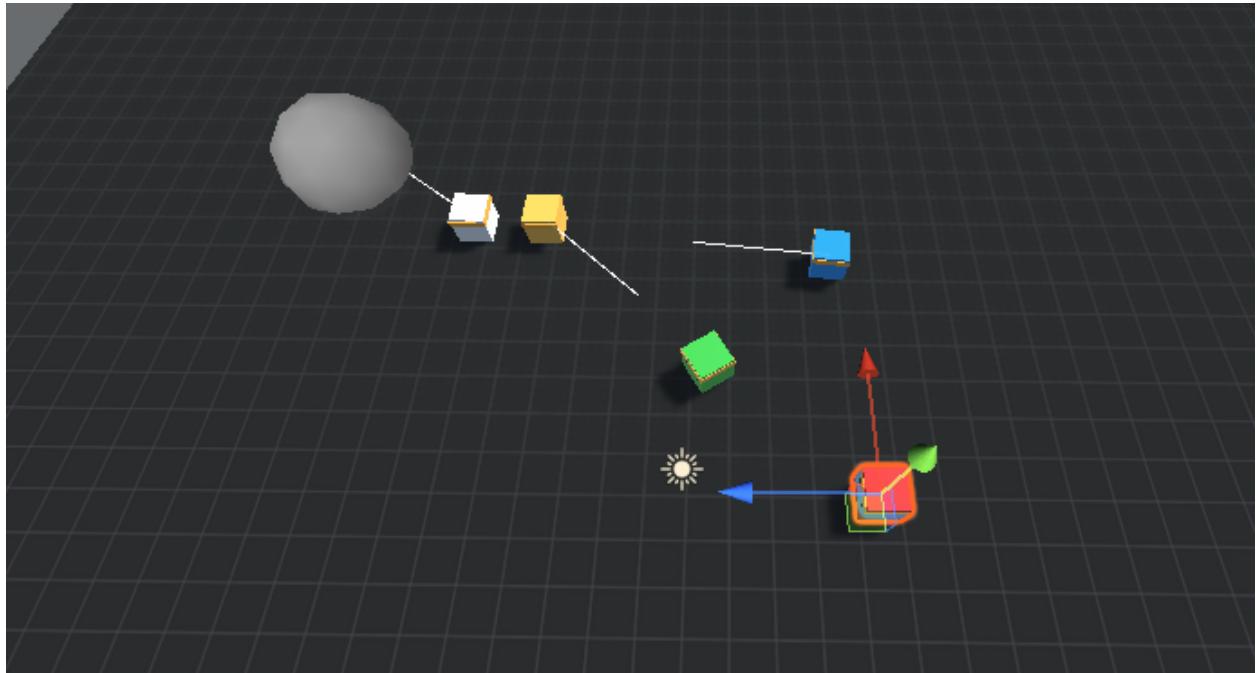


Figura 9. El agente amarillo implementa Persue, el blanco Evade, el verde Face y el azul Wander.

Wander:

Crea una posición aleatoria a en una área limitada por la posición del personaje y donde este mirando, posteriormente intenta llegar a la posición. Pasado un tiempo o si el npc ha llegado al a la posición crea otro punto y se dirige a él.

Path Following (sin Path Offset)

Este comportamiento usa dos scripts. El resultado es que un agenteNPC puede seguir a otro.

Para usarlo tenemos que asignar al objeto a seguir el script *Patheable*. Para que el otro agente lo siga necesitamos el camino recorrido. Este script guarda el path creando una lista de nodos. De esta forma guardamos las posiciones por las que ha ido pasando el objeto.

Al objeto que sigue el path le ponemos el steering *followPath* que hace un *Seek* a los nodos.

**e) Hacer detección de colisiones sobre obstáculos y objetos móviles usando solo WallAvoidance. Deberá trabajar con 1, 2 o 3 bigotes.**

Inicialmente nuestro proyecto usaba 1 único bigote. Aparecieron demasiadas colisiones ya que uno solo no es suficiente para detectar obstáculos correctamente. Acabamos implementando también con 2 bigotes, aunque este sigue representando problemas en las esquinas o paredes obtusas.

Wall Avoidance (1 bigote):

El personaje utiliza un raycast (lanza un rayo) al que llamamos bigote que detecta cuando hay una colisión a una distancia determinada por la variable *lookAhead*. Si ese raycast choca contra una pared y hay colisión, se calcula un punto a una distancia determinada por el valor de *avoidDistance* en el cual sepamos que no vamos a chocar contra el muro.

Dicha posición es calculada como el punto donde chocó el rayo sumado a la normal de dicho punto y multiplicado por *avoidDistance*. Una vez tenemos dicha posición hacemos un *Seek* a dicha posición, evitando así el muro.

Wall Avoidance (2 bigotes):

En este caso utilizamos dos bigotes separados entre sí por un ángulo determinado por la variable *angle* y con distancias de los rayos determinadas por dos variables distintas. Si el bigote izquierdo o el derecho nota colisión entonces giramos para evitar el choque. Ante la situación de que choquen los dos bigotes a la vez hacemos caso al bigote izquierdo, ignorando el derecho en este caso.

Mostramos su uso en la Figura 10.

En la Figura 11 vemos como no es infalible.

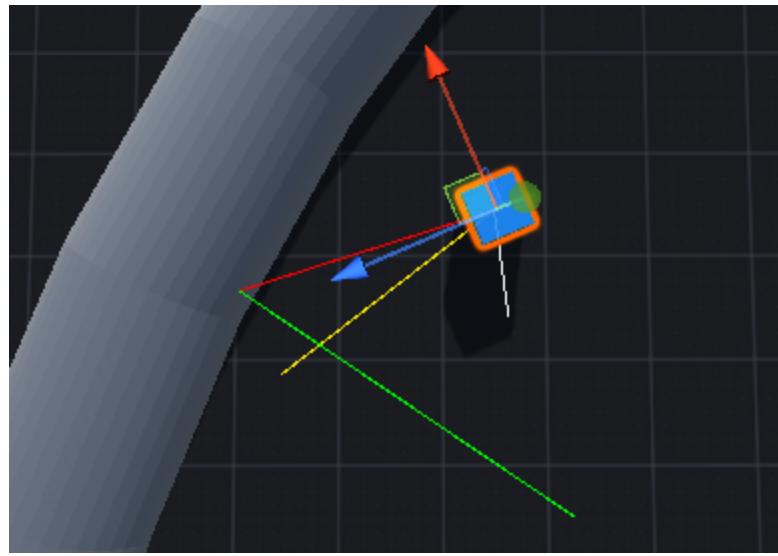


Figura 10. Tenemos dos bigotes. El rojo ha detectado colisión y el vector calculado al que se dirige el agente es el verde.

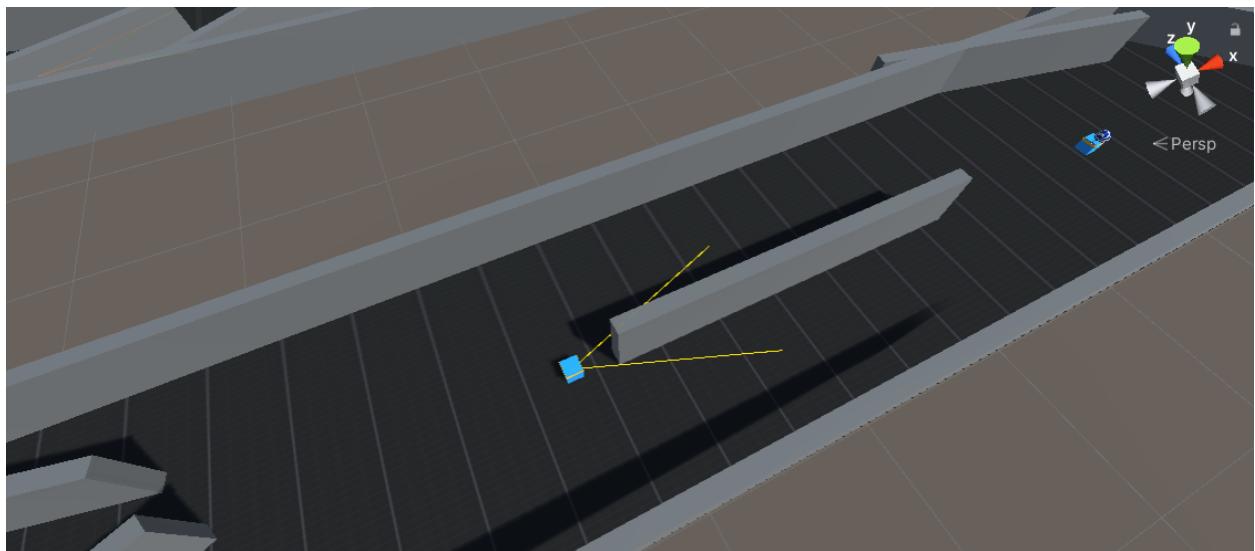


Figura 11. Presenta un problema si la pared viene de frente o es obtusa(<).

- f) Formaciones: Los personajes seleccionados pueden recibir la orden de realizar una formación. Se usarán al menos dos estructuras fijas preestablecidas (defina las estructuras claramente en la documentación).

Lo hacemos usando 2 clases Formation y FormationOffset además de varios añadir varios estados internos al Agente NPC. Vemos un diagrama en la Figura 12.

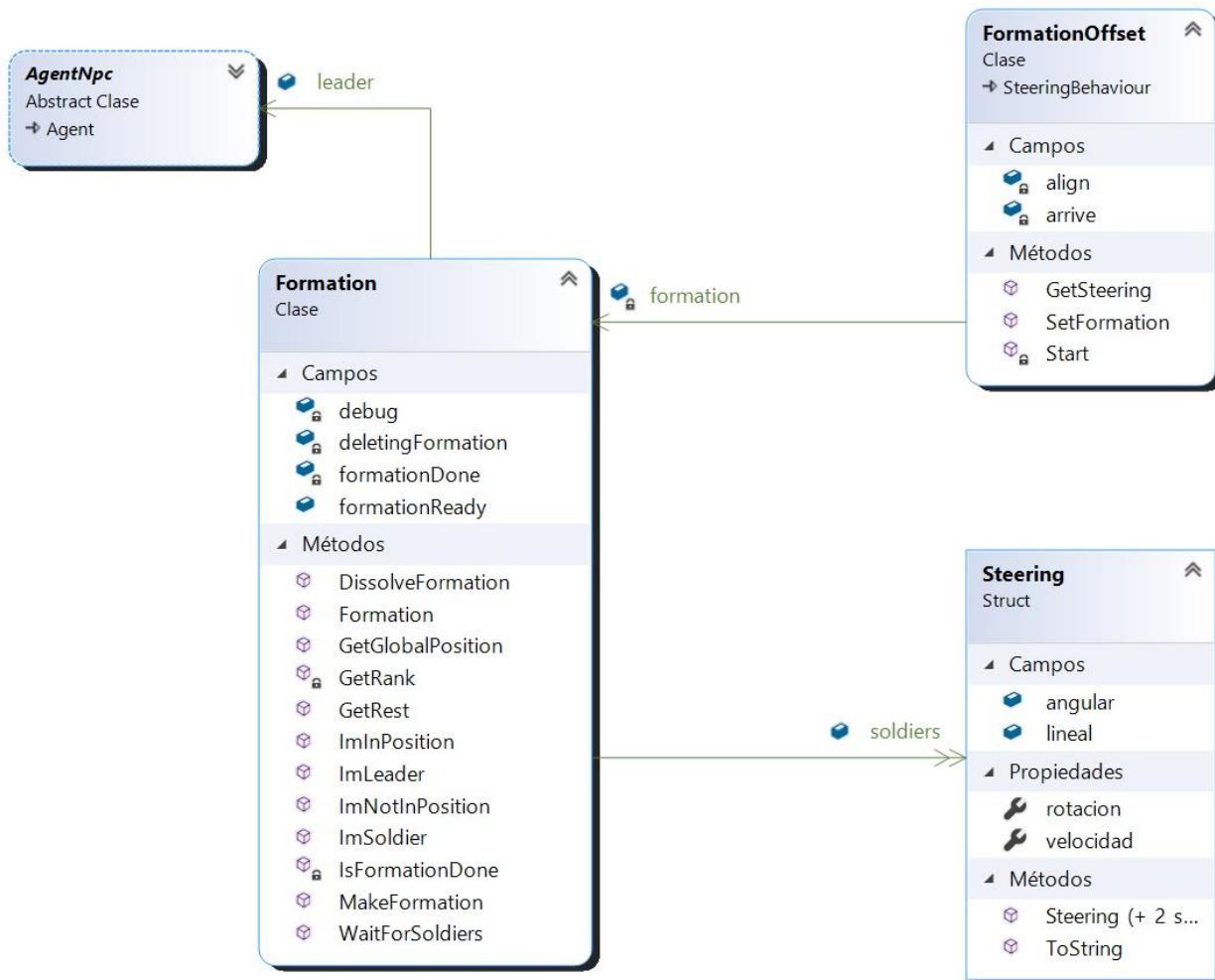


Figura 12. Clases usadas para la formación.

*Formation* es la clase que lleva el estado interno de la formación, también se compone de un líder (agenteNpc) y un keymap de Soldado(agenteNpc) y steering que es la posición local con respecto al líder. También incluye varias funciones para obtener información sobre la formación.

La clase *formation* permite que el controlador la cree y asigne las posiciones a cada uno, finalmente se llama a *MakeFormation* que cambia el estado y la acción de los agentes npc para que luego sus árbitros tengan en cuenta el steering de *FormationOffset*

*FormationOffset* es el steering que calcula dónde tiene que ir cada soldado (el líder sigue sus propios steerings), para esto se basa de la clase Formation que lleva un keymap con cada soldado y su posición local respecto al líder.

Las dos posibles formaciones son en forma de línea o de x, se muestra en la Figura 13 y 14.

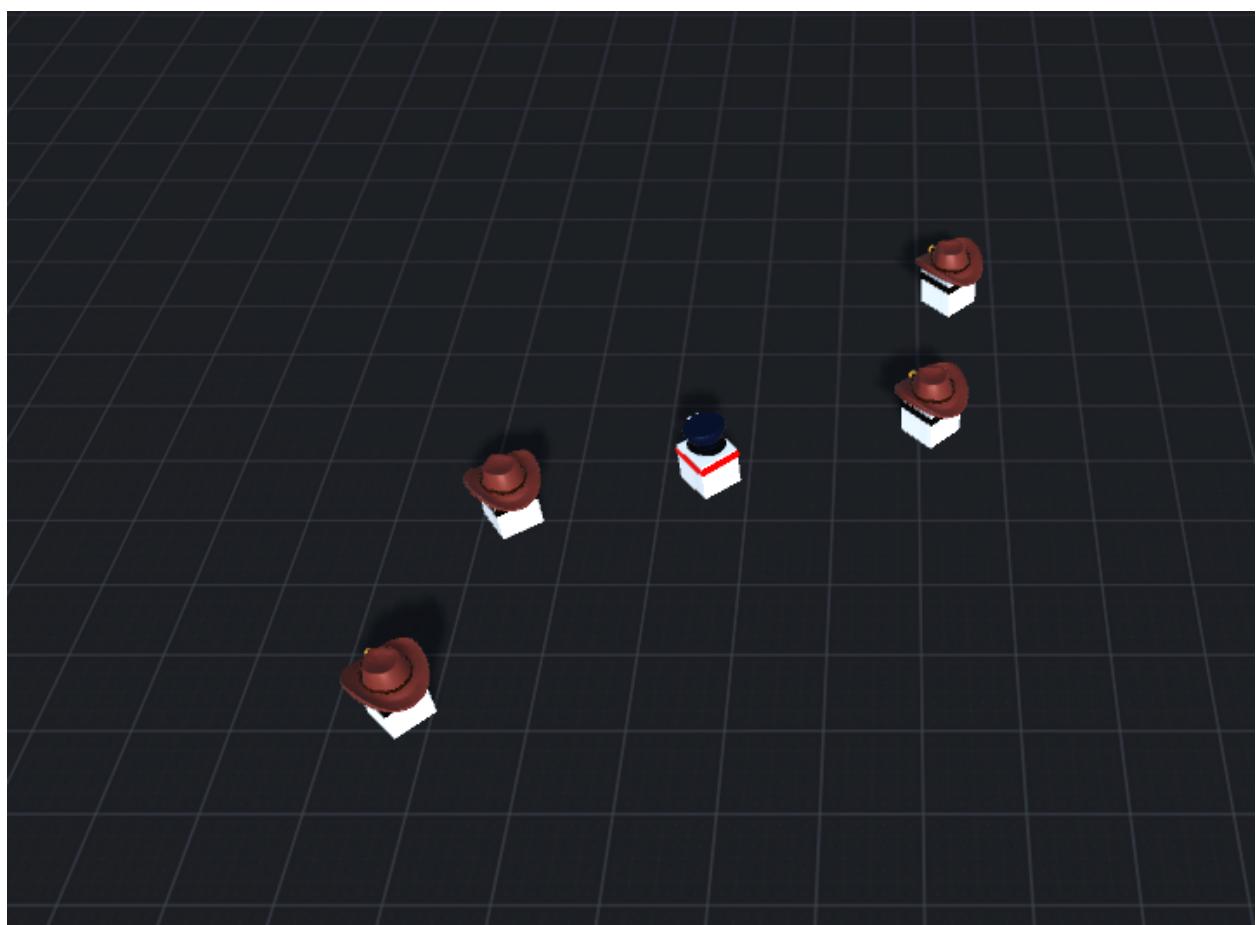


Figura 13. Formando una formación en línea.

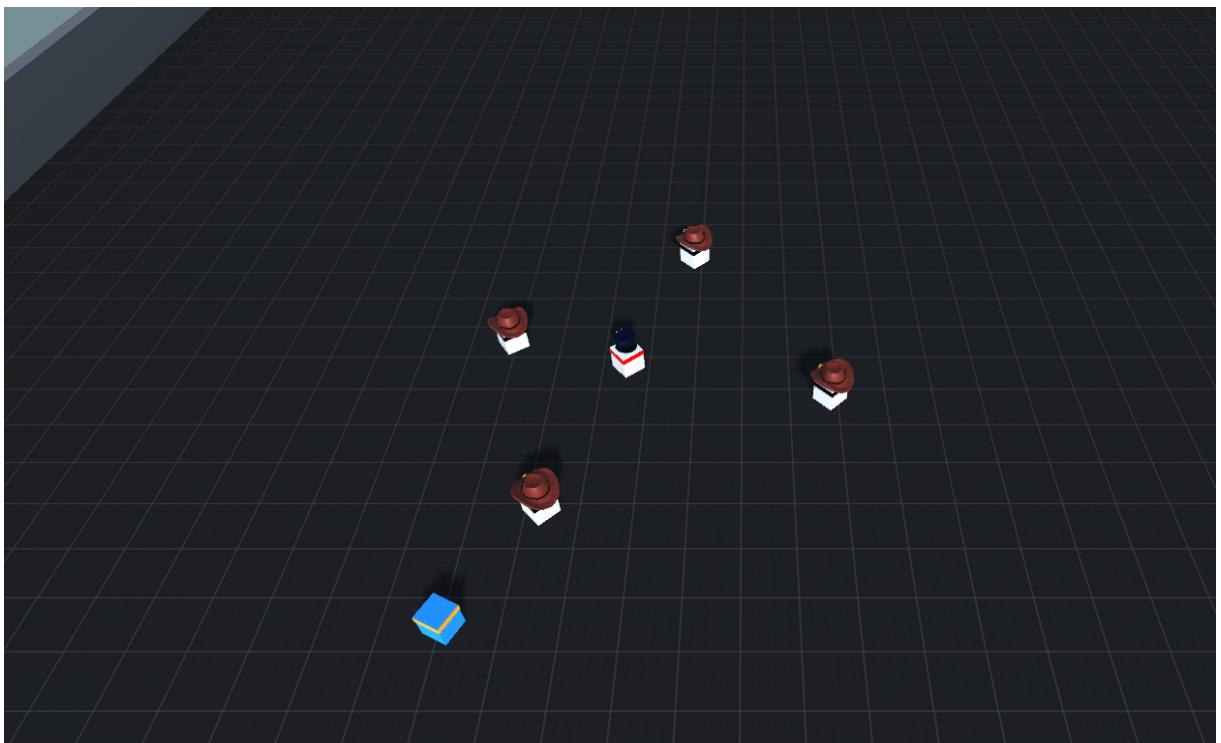


Figura 14. Formación en X con el líder en el centro.

- g) Contendrá personajes capaces de tener comportamientos de steering de grupo, al menos, flocking. (pájaros, peces del río, ...) realizando, simultáneamente, persecución a un personaje Wander.

Lo hacemos con los tres steerings de grupo. Estos steerings permiten a los agentes obtener información de los personajes que tienen alrededor y dependiendo de si están cerca o lejos hacen un movimiento u otro.

Los tres movimientos son Alignment, Cohesion y Separation.

- Alignment hace que todos el grupo mire a un lugar común.
- Cohesion hace que el grupo vaya a un centro común y se juntan.
- Separation es contrario a Cohesion. Si se acercan mucho se separan.

La escena Flocking muestra este comportamiento. Vemos un ejemplo en la Figura 15.

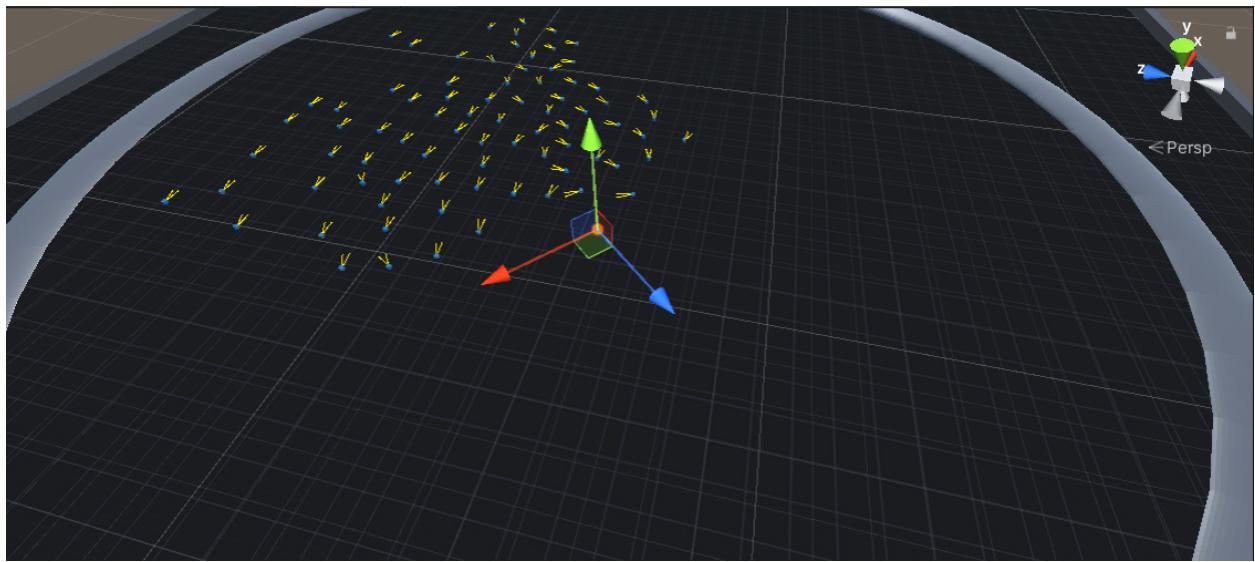


Figura 15. los agentesNPC siguen en grupo al objeto seleccionado

**h) Crear, al menos, un árbitro (basado en ponderaciones) para cada tipo de personaje.**

El árbitro que usamos con pesos es *BlenderSteering*. El árbitro tiene asignados varios steerings, cada uno con un peso y produce un steering final que es la combinación de todos.

Por ejemplo, si nuestro personaje tiene un steering de peso 1 que indica un movimiento a la izquierda, y un steering de peso 2 que indica dos movimientos a la derecha (equivalente a cuatro movimientos a la derecha al tener peso 2), la combinación de ambos steerings que hará el árbitro será tres movimientos a la derecha.

Nuestro proyecto incluye otro árbitro adicional como parte de los elementos opcionales. En dicho apartado veremos ambos también de forma gráfica.

**i) Se implementará el LRTA\* con espacio de búsqueda local para el subespacio minimal.**

Una de las funcionalidades implementadas en nuestro proyecto es la búsqueda del camino más corto para llegar a un objetivo (*PathFinding*).

Para la búsqueda necesitamos un grid del tablero, la heurística que aproxima la solución al objetivo final y un objetivo. Para implementarla tenemos cuatro clases. El algoritmo LRTA es una de ellas.

- GridChungo: Para calcular el path a un objetivo necesitamos un tablero con la posición del personaje, la posición del objetivo y las casillas que podemos visitar. Esta clase crea un tablero donde cada posición refleja un área en el espacio. El tablero lo representamos como un array bidimensional de nodos. Cada casilla del tablero es por tanto un nodo.
- Nodo: Un nodo guarda su posición en el tablero, representada con puntos x e y, la posición que tiene en el mundo real esa celda del tablero, si se puede visitar o no, y su coste. El coste de un nodo está formado por el coste de la heurística(h) y el coste real usado para ir a este nodo(g). El coste lo usaremos en el algoritmo LRTA.
- LRTA: Es la clase que implementa el algoritmo de búsqueda. La versión que usamos es LRTA con look-ahead de 1.
- PathFinding: Usaremos la tabla siguiente para mostrar los pasos que sigue el pathFinding.

	C g=∞, h=∞	A g = 0, h=2	D g=∞, h=∞	
	B g=2, h = 1	F g=3, h=4	E g=∞, h=∞	
X				

En la tabla comenzamos en el nodo A, que tiene un coste g de 0 y un coste h de 2. El objetivo se encuentra en el nodo marcado con X.

1. El primer paso consiste en poner el nodo actual en el nodo inicial e inicializar la lista de nodos y las variables necesarias.
2. Calculamos el coste para cada vecino de alrededor. Normalmente tendremos 8 casillas alrededor de nosotros (si no estamos en una esquina ni pegados a un objeto como en este ejemplo donde tenemos 5 vecinos). Nos encontramos en A y tenemos un coste de 2 (coste real g + coste heurística h).

3. Elegimos la casilla vecina de menor coste. En el ejemplo elegimos B.
4. Comprobamos si el coste del vecino elegido es mayor que la heurística de la casilla anterior. Si es así actualizamos el coste de la casilla anterior (será la inicial si es el primer movimiento).

En el ejemplo hemos elegido B y tenemos un coste ( $g+h$ ) de 3. Al ser mayor que el coste de heurística del estado inicial actualizamos A con 3.

5. Nos movemos a B y vamos a paso 2.

Si en algún momento el coste de ir hacia delante es más caro que ir hacia atrás retrocedemos y buscamos otro camino. Si no hay camino damos vueltas infinitas.

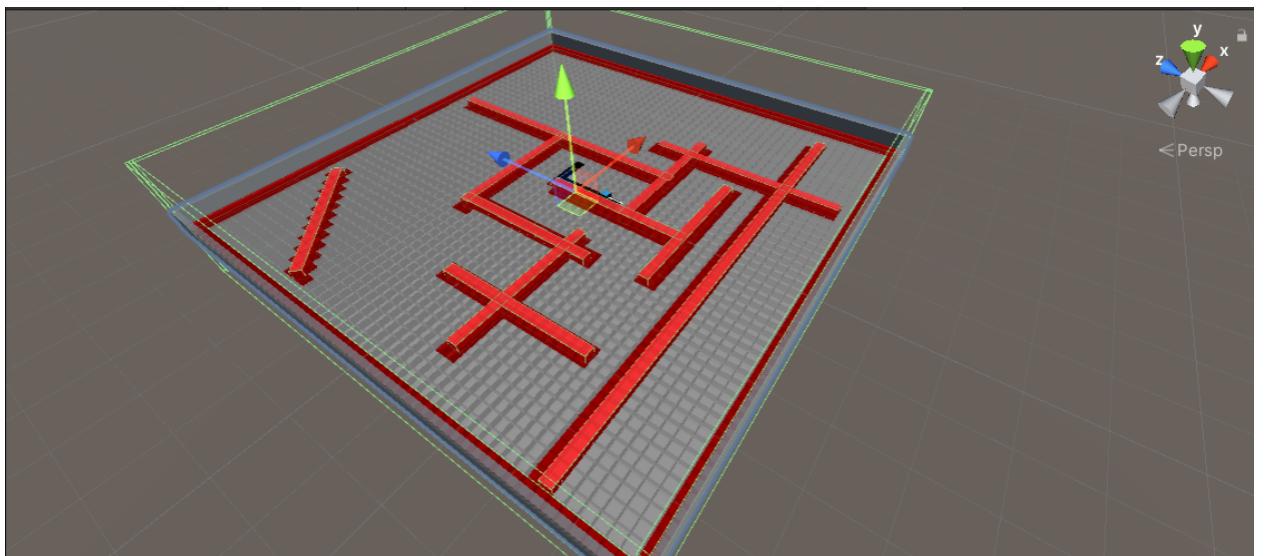


Figura 16. ejemplo LRTA con heuristica de chevishev.

- j) **Modo depuración.** Si se selecciona este modo, se mostrarán las figuras envolventes de los personajes y obstáculos, vectores actuales de aceleración, aceleraciones deseadas, etc ...

La opción depuración se controla con una variable en el método *OnDrawGizmos()*. Podemos activar la opción debug en el mismo inspector de Unity. Lo tenemos en todos los agentes y steerings.

Con esta opción obtuvimos las Figuras 2 y 3 mostradas anteriormente. Si lo usamos con un steering, e.g. Interpose, podemos ver en la Figura 17 como calcula el punto medio entre los objetos.

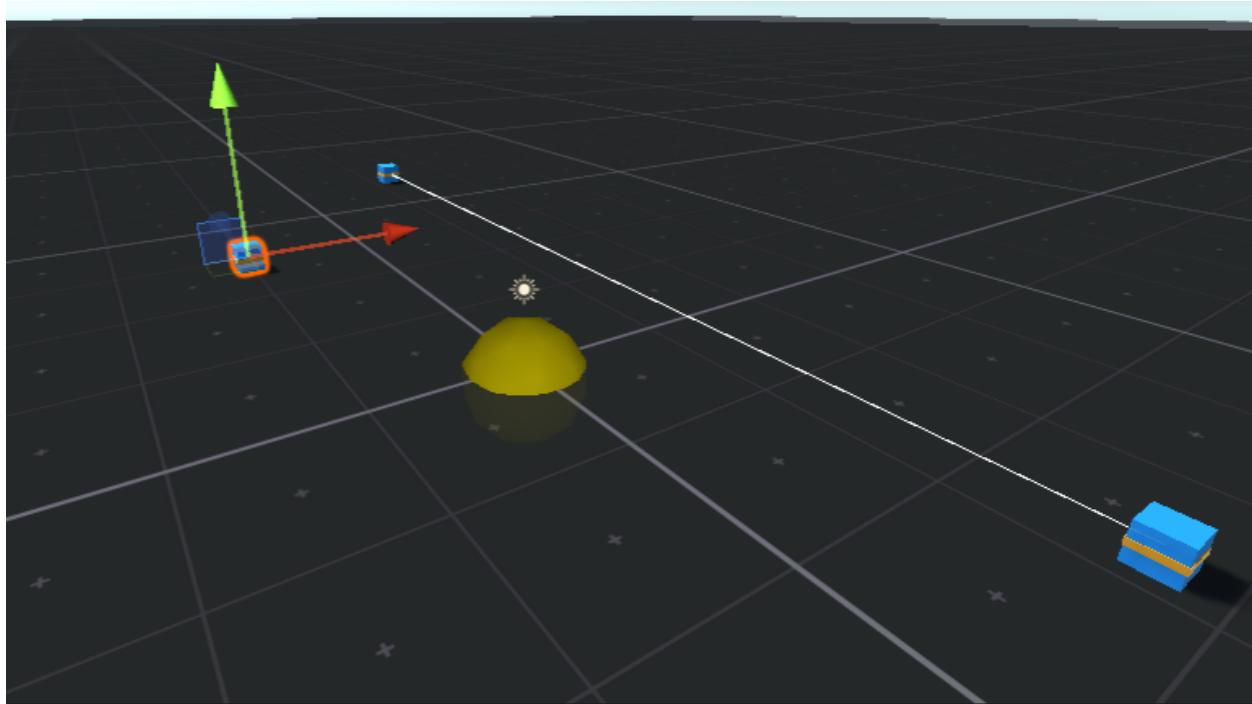


Figura 17. La opción debug usada en el steering Interpose.

También tenemos elementos opcionales.

a) Implementación del siguiente paquete de steerings.

**Collision Avoidance basado en predicción, PathFollowing con offset predictivo, Interpose, Wall Following.**

Opcionalmente puede añadir a cada paquete Seek, Free, Arrive, Wander basado en velocidad, LookWhereYouGoing, Containment, Flow Field Following, ....

Look where you going:

Este steering gira el NPC para que el personaje mire de frente hacia donde está viajando. Esto se consigue obteniendo el ángulo entre su vector velocidad y mi vector *forward* (vector hacia donde está mirando el personaje actualmente) sabiendo este ángulo ya podemos hacer un *Align*.



### Interpose:

Dados dos objetivos, este steering busca interponerse entre los mismos posicionándose en mitad del camino. Para ello, primero se calcula el punto medio entre los dos objetivos sumando sus posiciones y dividiendo por la mitad. Teniendo el punto medio, calculamos el tiempo que nos tomaría llegar a dicho punto para predecir a futuro dónde estarían los objetivos en ese tiempo, con el fin de ir al punto medio de esas futuras posiciones llamando a un *Arrive* para llegar a ese futuro punto medio.

### Path Following (con Offset predictivo):

Permite la misma funcionalidad que el Path Following explicado anteriormente. La diferencia es que en este caso predice la posición futura de los nodos. Calculamos la distancia con la posición futura en vez de con la posición actual.

### Wall Following:

Tenemos dos tipos de Wall Following:

- *WallFollowingCircular*

Se mantiene a una distancia constante siguiendo el exterior de un cilindro. El usado en la Figura 18a espera a que alguno de los bigotes colisione con el círculo y una vez que pasa podemos obtener la posición del centro del círculo del collider de Unity. Con esta posición calculamos el vector que va de nuestra posición actual al centro del círculo (el verde) y le aplicamos una rotación de 10 o -10 grados obtenemos el punto usando la ecuación de la recta y hacemos un *Seek* a ese punto.

Este steering lo hemos llamado *WallFollowingCircular* aunque se parece más a un *Containment*.

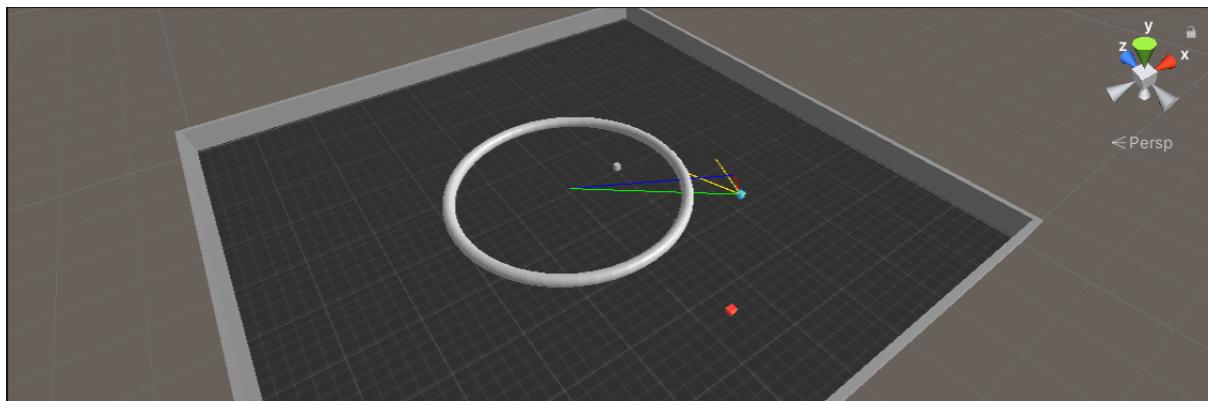


Figura 18a. WallFollowing en una escena circular.

- *WallFollowing*

Nos basamos en *WallAvoidance* con dos bigotes pero en vez de tener que poner la longitud, ángulo y separación de la pared usamos unos bigotes de 45 grados y gracias a pitágoras sabiendo solo la longitud que queremos separarnos de la pared podemos calcular el resto de parámetros.

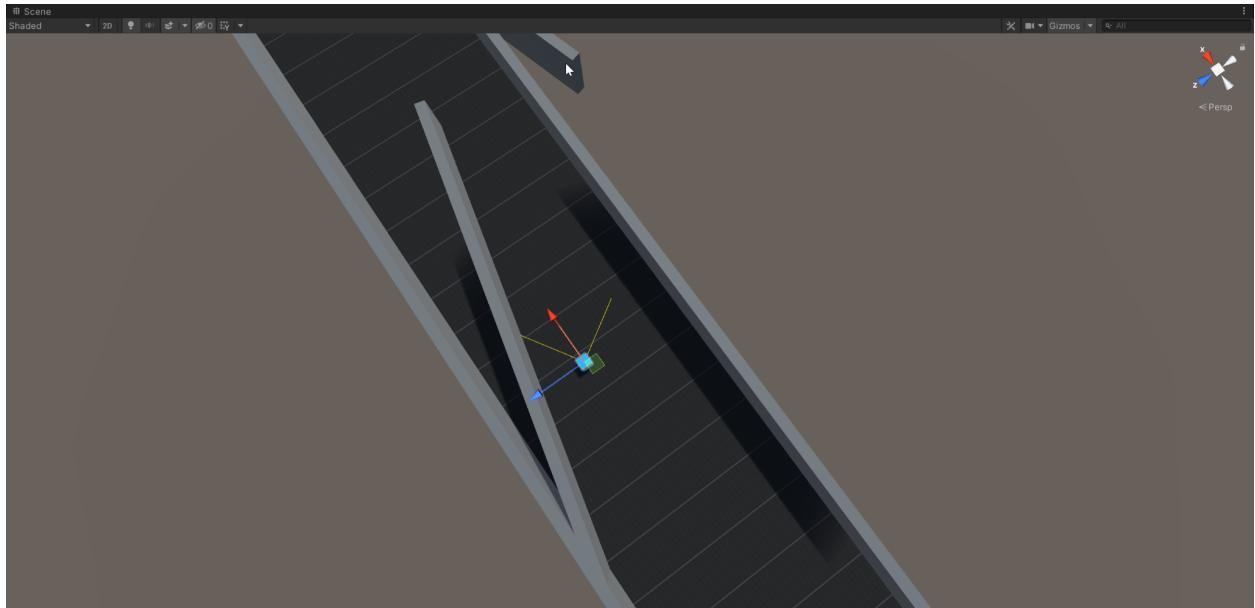


Figura 18b. WallFollowing en una escena circular.

#### Collision Avoidance (basado en predicción):

Collision Avoidance busca en cada momento el personaje más cercano con el que vamos a colisionar calculando el tiempo de respuesta de colisión y quedándonos con el más pequeño,

ignorando a aquellos personajes cuya distancia de separación sea mayor que 2 veces nuestro rango exterior.

Si la separación que obtenemos de nuestra colisión más cercana es muy pequeña significa que estamos actualmente chocando, por lo que intentamos apartarnos.

Si por el contrario es mayor que dos veces nuestro radio exterior, significa que podemos predecir el punto de choque, por lo que utilizamos la posición relativa en su lugar como punto al que viajar para evitar la colisión.

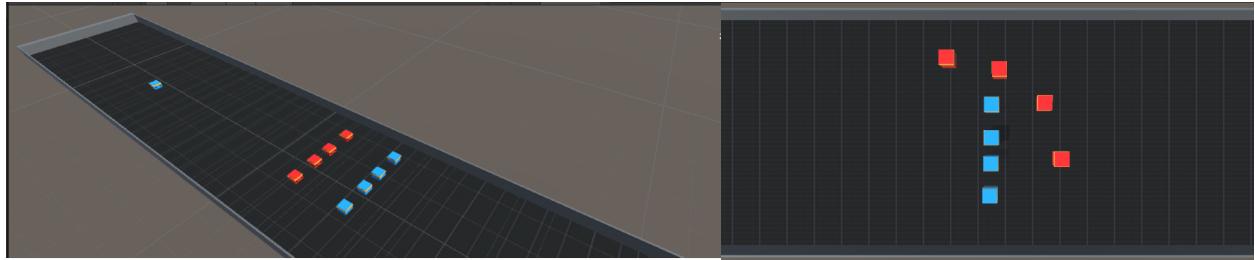


Figura 19. En la izquierda vemos la posición inicial. En la derecha ambos bandos se dirigen a sus objetivos.

### **b) Usar la arquitectura de arbitraje (steering combinado) basada en prioridades dinámicas.**

El otro tipo de árbitro que hemos implementado es *PrioritySteering*. A diferencia de *BlenderSteering* que utiliza pesos, este tiene una lista de prioridades. El orden seguido por defecto es Collision, Formation, Pursuit y Distance.

Si nuestro agente tiene asignado este árbitro y dos steerings, uno de colisión como *WallAvoidance*, y otro de persecución como *Seek*, primero calculará la colisión de forma que cuando se vaya a producir devuelve un steering a 0. Cuando no colisiona pasa al siguiente grupo que sería el *Seek*.

Si activamos la opción de orden mayor invertimos el orden de prioridades y el *Seek* tendría mayor importancia. Si hubiera una pared siempre se chocaría.

Para implementar este árbitro internamente usamos el árbitro con pesos. Cogemos el primer grupo (Colisión) y calculamos el steering final con *BlenderSteering*. Definimos un valor  $\epsilon_{mu}$  muy pequeño de forma que si la magnitud es menor que este valor pasamos al siguiente grupo, sino devolvemos este y no calculamos el resto.

En la Figura 20 podemos ver en funcionamiento ambos árbitros. El agente blanco tiene el *BlenderSteering*, el marrón el *PrioritySteering* con las prioridades por defecto y el amarillo con las prioridades invertidas. Los agentes tienen el steering de colisión *WallAvoidance* y de persecución *Seek*, para ir a los agentes verdes que permanecen quietos.

En el comportamiento resultante vemos que el agente blanco y el marrón consiguen pasar la pared. El blanco lo hace con un movimiento combinado de los dos steerings, el marrón usando únicamente el steering de colisión. Cuando esquivan la pared ambos se dirigen al agente verde.

Por el contrario, el agenteNPC amarillo no consigue esquivar la pared. Únicamente hace caso al seek que le indica la dirección del agente verde.

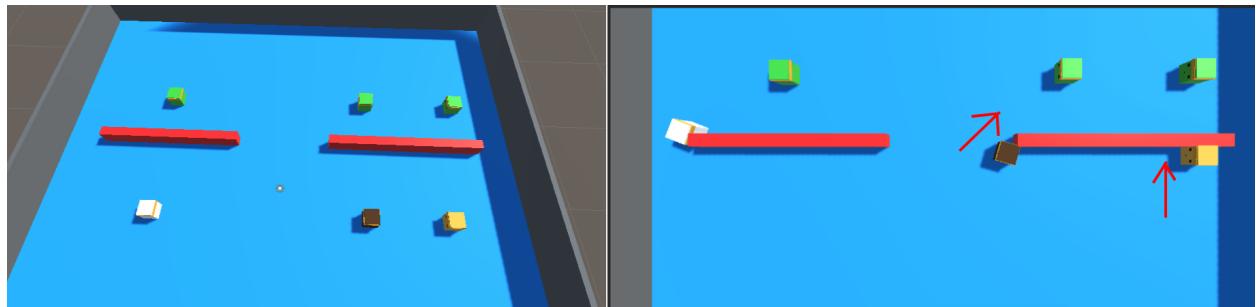


Figura 20. La escena PrioritySteering muestra agentesNPC con los dos tipos de árbitros asignados.

### c) Construcción de dos actuadores con situaciones que claramente pongan de manifiesto la heurística usada.

Como vimos al inicio de la sección en el diagrama de clases encontramos un *BaseActuator* asociado al agenteNPC.

El *BaseActuator* es la clase de la que heredan los dos tipos de actuadores implementados, *FilterActuator* y *TankActuator*.

*FilterActuator* se encarga de eliminar la variable Y porque nuestros personajes no saltan. Es el actuador por defecto y además lo implementan internamente varios steerings.

*TankActuator* es el otro tipo de actuador. Lo usaremos con el agenteNPC de tipo Tanque. Se encarga de filtrar el steering recibido de forma que antes de hacer un movimiento lineal comprueba si tiene que hacer una rotación.

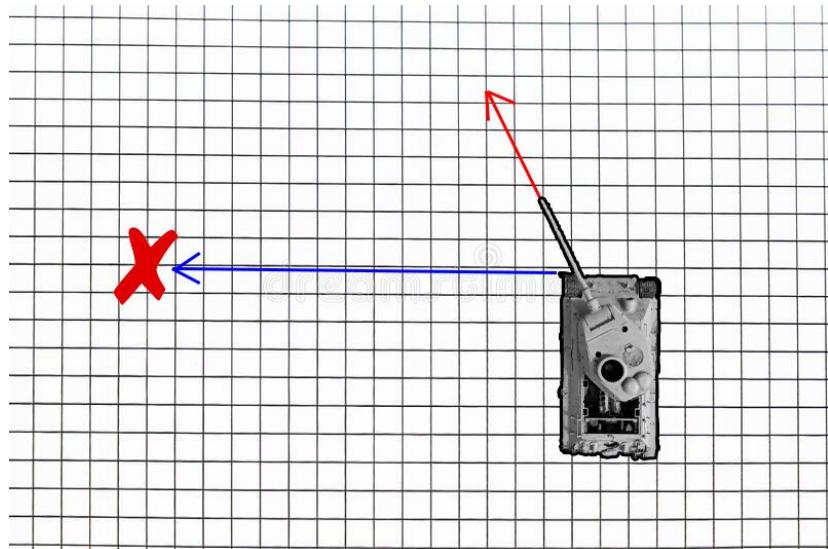


Figura 21. Interacción entre steering, árbitro, agenteNPC y un actuador.

Podemos usar la Figura 21 para entender su funcionamiento. Inicialmente, nuestro tanque se encuentra mirando en la dirección del vector rojo y queremos ir a la posición marcada con una X. El vector de movimiento resultante es el azul.

El actuador se encarga de obtener el ángulo formado por los dos vectores y en caso de ser distinto (realmente no comprobamos si el ángulo es 0, dejamos un pequeño error de 5 grados) devuelve al agenteNPC un steering con la rotación a realizar.

El agenteNPC realizará el movimiento a la posición señalada esté mirando a ella. De esta manera hacemos que nuestro personaje se mueva únicamente hacia delante y hacia atrás.

Podemos ver su uso en la escena *ActuatorScene* donde encontramos un tanque y dos arqueros con este filtro y con el steering *Seek* para seguir al agentPlayer controlado por el jugador.

## 3.2. BLOQUE 2: IA Estratégica y táctica

### Sistema de Combate

Para poder atacar nuestro personaje debe posicionarse a un máximo de *alcance* de distancia, donde *alcance* es un parámetro de cada agente para medir el alcance de cada tipo de unidad, teniendo un soldado distinto rango que un arquero y que un tanque. Para ello, utilizamos el gridmap cuadrático, donde calculamos si el número de casillas entre el personaje y el objetivo es menor o igual que el rango de nuestro NPC.

Nuestro sistema de combate calcula el ataque como el daño base del npc atacante, de su probabilidad de crítico y de si se encuentra en su mejor terreno. Todo esto depende del tipo de unidad, definiendo cada tipo sus parámetros. Así, los tanques tienen un 10% de probabilidad de crítico mientras que los arqueros tienen un 15%.

Cabe mencionar que, cuando realizamos el ataque, hacemos que el npc espere unos segundos (determinados por el tipo de unidad) antes de poder atacar de nuevo. Esto se consigue mediante la función *WaitForSeconds (secondsToWait)*, cuyo parámetro son los segundos en los que espera el NPC.

El cálculo de la defensa tan solo tiene en cuenta si estamos en nuestro mejor o nuestro peor terreno. En el primer caso multiplicamos en un factor de dos nuestra defensa base, mientras que en el segundo dividimos a la mitad. Finalmente, el daño resultante es el daño de ataque menos la defensa del objetivo.

### Escenario

c) Deben existir puntos seguros del mapa, o bien unidades especiales de curanderos, donde recuperar puntos de vida lentamente si la unidad permanece inmóvil.

Tenemos dos puntos seguros que son las fuentes de curación.

- d) Las unidades destruidas reaparecen en la base (quizá con algún retraso si eso agiliza el juego) y, en principio, acuden al lugar en el que fueron destruidas (un primer waypoint destino por defecto al reaparecer)

El sistema de combate tiene en cuenta diversos factores: el tipo de unidad, si estamos en nuestro mejor/peor terreno, la defensa del enemigo,

### Condiciones de Victoria

La condición de victoria de nuestro escenario general consta de destruir la base enemiga antes que el rival, es decir, el equipo cuya base tenga sus puntos de vida a 0 pierde. Cuando esto ocurre, se muestra un muñeco con el color del ganador, tal y como veremos más adelante en la sección de interfaz del usuario.

### Mapa Táctico

Se debe utilizar algún mapa táctico, como mínimo un mapa de influencia (es un elemento obligatorio). El mapa de influencia debería extenderse bastantes “casillas” alrededor de las unidades (como mínimo el rango de detección). La información sobre influencia debe incorporarse al pathfinding en este bloque 2.

Nuestro mapa de influencia trata de un mapa encasillado con divisiones cuadráticas que representa la influencia de cada equipo en el mapa.

Esta influencia viene dada por los objetos que tengan el script *IPropagator*, el cual se utiliza para identificar el objeto como un propagador de influencia, pudiendo ajustar además el número de casillas de distancia en las que influye mediante el parámetro *influencia*.

Un valor negativo de este valor indica que pertenece a un equipo, haciendo que se muestre en el mapa como influencia de color rojo, mientras que un valor positivo indica que pertenece al otro, mostrando un color azul. Si el valor de influencia es 0 entonces significa que ninguno de los dos

equipos tiene influencia en esa zona, o dicha influencia es neutra, mostrándose en el mapa con un color de casilla blanco.

Mediante el uso del script controlador del mapa llamado *InfluenceMapControl*, así como la estructura del mapa generada por el script *Influence Grid*, este va propagando sobre el mapa la influencia de cada objeto con el script *IPropagator*.

Para ello, cada cierto tiempo se llama a la función que se encarga de propagar sobre el mapa, haciendo que para cada propagador se actualice el nodo que está en su posición actual, así como los de su alrededor en función del radio exterior del propagador.

Una vez hecho esto, para cada propagador se actualiza el valor de todas las casillas a *influencia* posiciones de distancia desde arriba, abajo, izquierda y derecha, sumando a cada casilla el valor de influencia del propagador.

En el escenario *General* podemos ver un ejemplo del comportamiento del mapa donde, si activamos el modo de depuración del mismo seleccionando el objeto *InfluenceController* y dentro del script *Influence Grid* debemos activar la opción de *Mostrar Mapa*. Una vez hecho esto se dibujara sobre el escenario el mapa de influencia, tal y como vemos en la Figura 22.

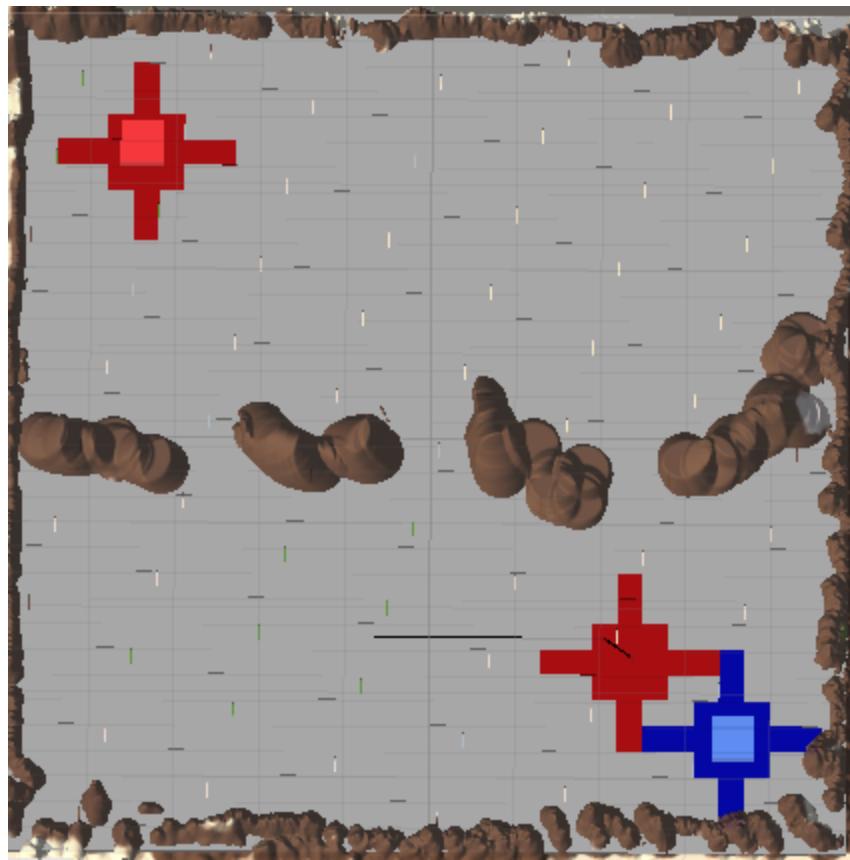


Figura 22. Mapa de influencia con dos bases y un soldado.

La información del mapa de influencia es utilizada tanto para el cálculo de los pesos del algoritmo A\* como para determinar si atacar a los puntos de interés en la toma de decisiones. En el primer caso, hacemos que será más fácil caminar por territorio aliado que enemigo, mientras que en el segundo caso se comprueba que puntos de interés faltan por dominar (aquellos con valor 0 o signo contrario al de nuestra influencia) para ir al primero de ellos.

### Comportamiento de los agentes

El comportamiento de los agentes viene determinado por tres modos: Guerra Total, modo de ataque y modo de defensa. Estos tres tipos de modos son implementados mediante los distintos Behaviour Trees de los diferentes tipos de agentes en función del comportamiento de cada tipo de NPC.

Nuestro comportamiento táctico viene dado por la toma de decisiones. Esto ha sido implementado con Behaviour Trees ya implementados y disponibles para nuestro motor de juegos [5].

Cada tipo de unidad tiene un árbol distinto y, por tanto, un comportamiento diferente al resto. Un ejemplo de esto es la función de huir a un punto curativo, donde dependiendo de la unidad es en una situación u otra.

En el caso de los soldados sólo huyen si tienen la mitad de la vida y están en modo de ataque. En los tanques, sólo en modo de defensa. Los arqueros huyen siempre que tengan menos de la mitad de vida o tengan a su alrededor a más enemigos que aliados.

El posicionamiento de estas condiciones de huida ya hace que cada tipo de NPC tenga comportamientos distintos, pero además queremos recalcar algunos comportamientos exclusivos de cada uno.

En el caso del tanque podemos observar su árbol de comportamiento en la Figura 27 de la sección 5, donde destacamos los detalles que separan con el resto de agentes:

- Los agentes no van a atacar a la base enemiga. Hemos pensado que sería mejor que defiendan los puntos de interés, teniendo así mayor cercanía a la base aliada en caso de una retirada para defender la base.
- Cuando activamos el modo Guerra Total, son los únicos que se quedan a defender la base, ya que el resto de agentes se dedica a atacar la base enemiga.
- En el modo de ataque se dedican a capturar puntos de interés.
- En el modo de defensa se dedican a defender la base aliada de enemigos.
- Sólo huirá a un punto curativo si su salud está por debajo de la mitad y se encuentra en el modo defensivo.

Si bien vemos que el tanque se trata de una unidad defensiva, podemos apreciar como los arqueros se dedican más al ataque, viendo que se caracterizan en:

- En el modo Guerra Total va directo a atacar a los enemigos e ir directo a la base enemiga
- Siempre que su salud disminuya por debajo de la mitad o existan a su alrededor más enemigos que aliados huirá a una fuente curativa, dado que estas zonas se consideran puntos seguros.
- En el modo de ataque se dedica a atacar enemigos y conquistar puntos de interés. Si todos los puntos de interés han sido conquistados en su lugar irá a por la base enemiga.

- En el modo defensivo irá a la base aliada a defenderla de los enemigos.

Examinando estas dos unidades vemos que ya existe un comportamiento distinto entre unidades, aunque vamos a mencionar también las características de los soldados, donde podemos ver su árbol en la Figura 29 de la sección 5:

- En el modo Guerra Total atacará a los enemigos cercanos e irá directo a la base enemiga.
- En el modo de ataque no conquista puntos de interés, sino que va a la base enemiga directamente, atacando a los enemigos cercanos y huyendo si se encuentra a menos de la mitad de la vida.
- En el modo de defensa se agrupa en la base aliada para defenderla.

Además, podemos ver en la Figura 30 de la sección 5 el árbol del tanque que se encarga de patrullar, tal y como se pide en el apartado a de los elementos opcionales, donde sólamente se dedica a ir viajando entre los dos puntos programados marcados por objetos vacíos con la etiqueta *Patrulla* y si encuentra un enemigo cerca lo ataca.

Por último, con el fin de completar el apartado opcional *d* hemos implementado diversos modos de información táctica a la hora de tener en cuenta las decisiones. Un ejemplo de esto son los arqueros, que tienen en cuenta la densidad de los enemigos con el fin de huir de una batalla en desventaja, y por otro lado el uso del mapa de influencia con el fin de conquistar puntos estratégicos.

## Guerra Total

**Por último, implementar un comando para activar “La guerra total”. Cuando se da esta orden, la IA toma el control de ambos bandos y los lanza a la batalla final (p.e. todas las unidades se ponen en modo agresivo y se busca muy activamente conseguir las condiciones de victoria). ¡Solo puede quedar uno!**

El controlador del escenario General es quien se encarga de activar el modo Guerra Total cuando la cámara detecta que se ha pulsado el botón de Guerra Total cuando estamos en el modo de juego de Unity, donde si activamos este modo hacemos que todos los NPC con un árbol de decisión activen dicho árbol, dándole así el control a la máquina del juego.



Además, hacemos que las bases activen dicho modo, haciendo que los árboles de decisión de cada npc tengan en cuenta este parámetro y busquen atacar a la base enemiga sin tener en cuenta nada más (o defender si es un tanque), de modo que el equipo que destruya la base enemiga más rápido gana.

Por último, para que se note de manera gráfica que nos encontramos en este modo, se muestra una ventana que indica que estamos en dicho modo.

### **Pathfinding Táctico**

Se basa en el uso del algoritmo A\* con una estructura tipo head con el fin de agilizar el algoritmo.

Para el cálculo de los caminos hemos utilizado el mapa de tipo grid con el fin de utilizar dicho mapa como un grafo sobre el que utilizar el algoritmo y, donde cada tipo de NPC utiliza un tipo de heurística diferente. Así, el soldado usa la heurística Euclídea, el arquero Chebychev y el tanque Manhattan.

Además, con el fin de completar el apartado opcional f, hemos implementado que el pathfinding tenga en cuenta dos modos de información táctica: por un lado tiene en cuenta la influencia enemiga como aliada, mientras que por otro lado se tiene en cuenta para los costes si nos encontramos en nuestro mejor o nuestro peor terreno.

## **4. INTERFAZ DE USUARIO**

### **Modos de comportamiento**

Para el cambio entre los distintos modos de comportamiento de los personajes (ataque, defensa y Guerra Total) hemos implementado unos botones que detecta la cámara en modo de juego de Unity cuando son presionados, cambiando el comportamiento y mostrándonos una ventana donde se indica el modo actual, tal y como se muestra en la Figura 23.

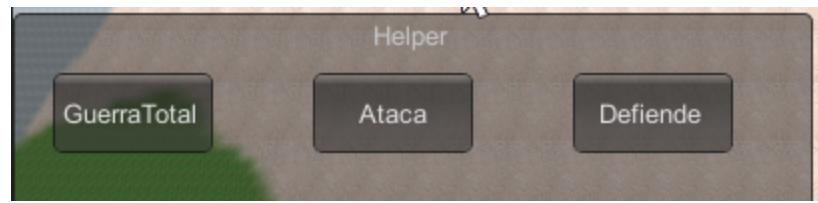


Figura 23: Botones de cambio de modo en el modo juego de Unity



Figura 24: Ejemplo de ventana que muestra el modo activado

## Pantalla de victoria

Cuando se cumple la condición de victoria (una de las bases baja su vida a 0), cuando gana el equipo rojo se muestra la Figura 25, mientras que para el equipo azul se muestra en la Figura 26.



Figura 25. Pantalla de victoria para el equipo rojo.

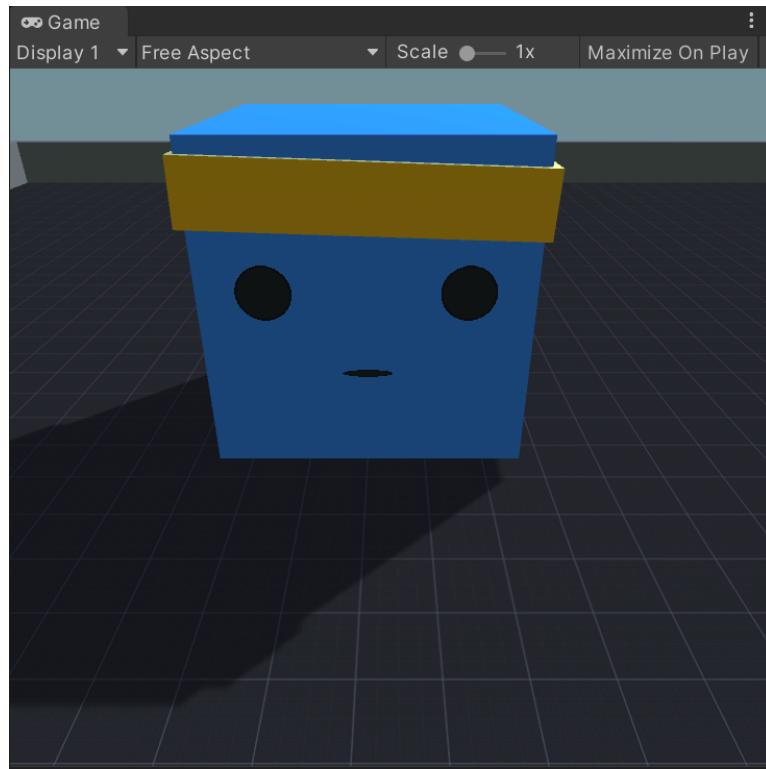


Figura 26. Pantalla de victoria para el equipo azul

## 5. DIAGRAMAS QUE REFLEJEN LA IA TÁCTICA IMPLEMENTADA

En las siguientes Figuras mostramos los diagrama en árbol usados en cada modelo.

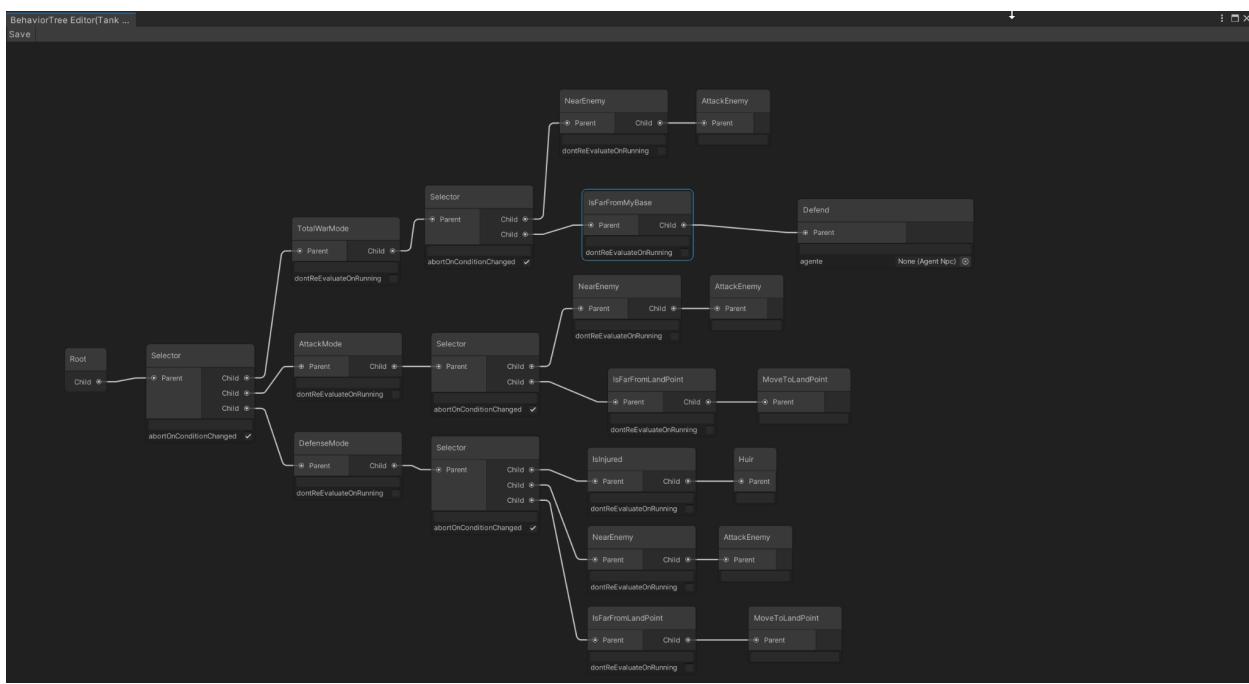


Figura 27. Diagrama para el modelo de tanque.

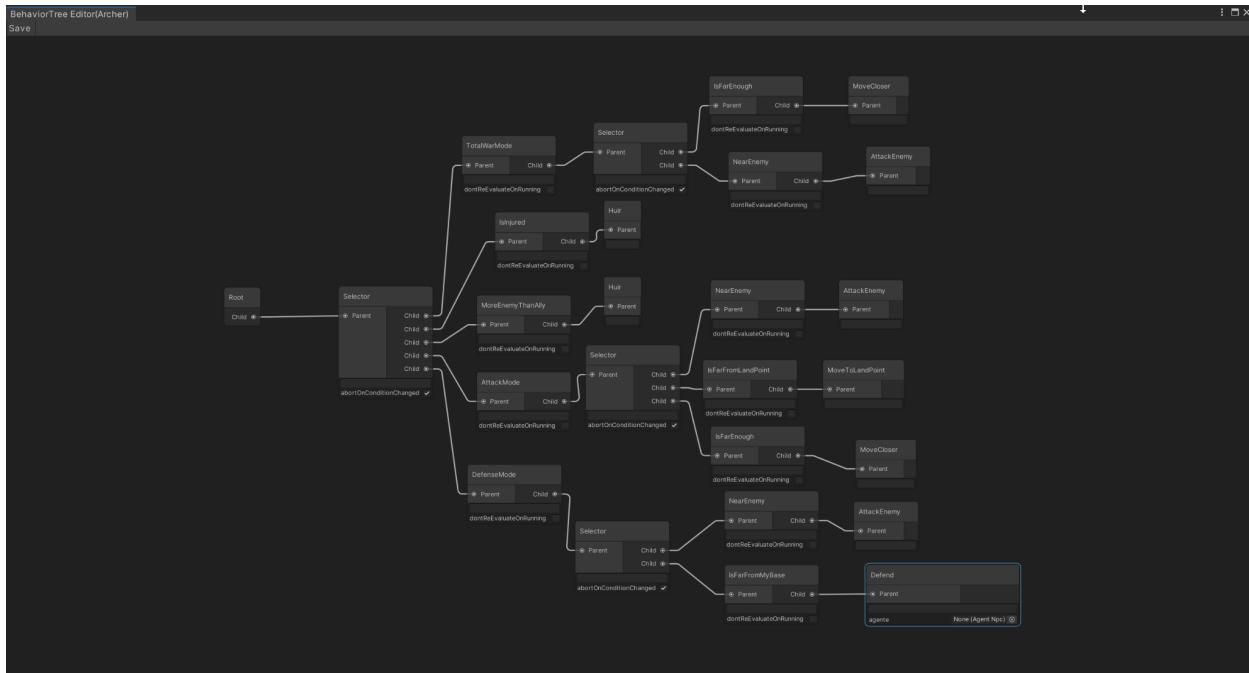


Figura 28. Diagrama para el modelo de arquero.

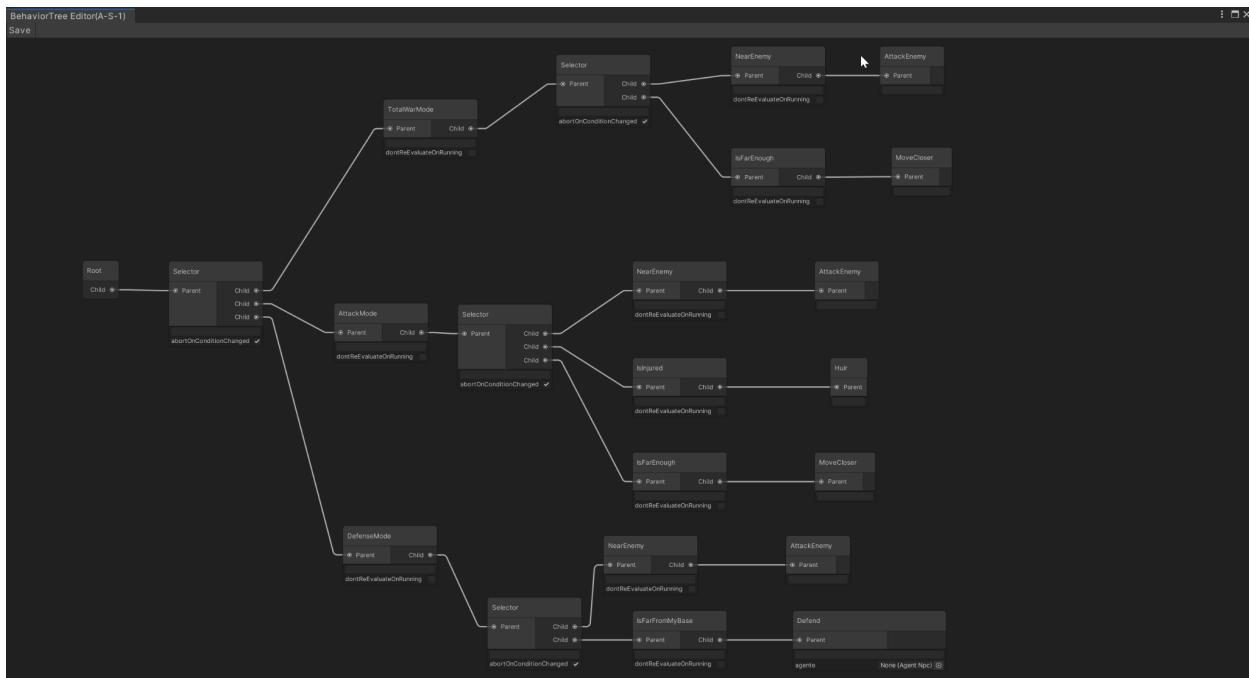


Figura 29. Diagrama para el modelo de soldado.

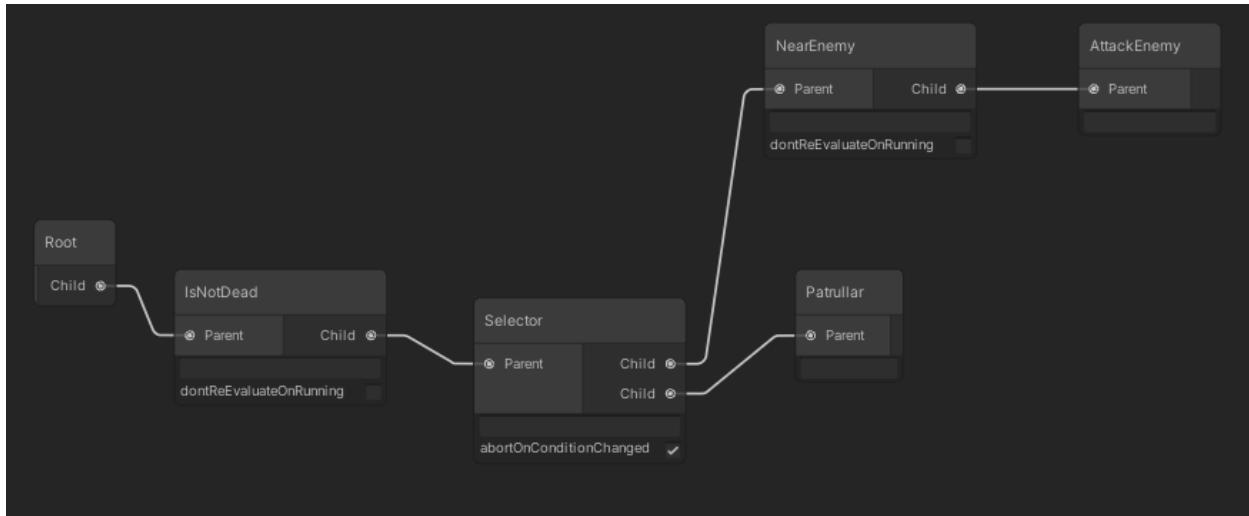


Figura 30. Diagrama para el modelo de patrullero.

## 6. GUÍA DE USO DEL PROYECTO

En esta sección indicaremos el manejo de los elementos que usan teclas y ratón y la funcionalidad de cada una de ellas.

## Formación

Comenzamos seleccionando los agentes con el ratón. A continuación:

- Tecla “l” para hacer una formación en forma de fila.
- Tecla “x” para hacer formación en forma de cruz.
- Tecla “r” para resetear los objetivos seleccionados con el ratón.
- Tecla “c” para deseleccionar los objetivos seleccionados con el ratón.
- Tecla “g” para mandar a un lugar marcado posteriormente con el ratón.

## 7. CONCLUSIONES

Esta práctica nos ha ayudado a mejorar nuestros conocimientos sobre Unity y dominar su uso. Durante el desarrollo, en especial de la primera parte, hemos adquirido conocimientos sobre el movimiento de los personajes en los videojuegos o la búsqueda de caminos.

En general, hemos conseguido entender la inteligencia artificial de otra manera: en el contexto de los videojuegos. Ahora sabemos lo que hay detrás de la “inteligencia” de los personajes y cuando veamos a uno moverse ante ciertas acciones para simular inteligencia veremos los videojuegos de forma distinta.

El desarrollo del proyecto ha supuesto una constante búsqueda de información sobre vectores e ideas para las físicas (un repaso general de cinemática) y distintas formas de implementar ciertas funciones de los scripts en C#, al tener menos experiencia con este lenguaje o no tener suficiente información, un ejemplo lo encontraríamos en las formaciones.

Una opción para mejorar el proyecto en el futuro sería indicar una estructura más clara para seguir, en concreto con los steerings delegados hemos tenido el problema de que implementar su target tal como en los ejemplos hacía que se moviera el personaje provocando errores.

Para solucionarlo, creamos un flag para no tener que comprobar que existiera un target y poder trabajar con vectores y puntos directamente.

Este problema fue aún mayor en el caso de las formaciones ya que no tenemos suficiente información para solucionar este desafío.

Otro problema que nos encontramos fue que al principio implementamos *WallAvoidance* con un único bigote y posteriormente tuvimos que cambiar a dos. El problema con uno solo es que hay varios momentos donde colisionó por no detectar bien la pared. Con dos bigotes también podríamos tener el problema si encontramos esquinas obtusas, pero no tenemos en nuestros mapas.

Para la IA hemos usado behaviour trees, decir que primero usamos el paquete de árboles que daban y luego encontramos el de las referencias

Como conclusión final, aunque una vez hecho el proyecto vemos que está entretenido, la práctica nos ha supuesto un gran esfuerzo ya que tiene demasiada carga de trabajo que hemos llevado en condiciones muy malas al mezclarse con exámenes de dos convocatorias distintas.

Esto nos ha llevado a implementar únicamente los elementos obligatorios mínimos, aunque siempre hemos mantenido la idea de hacer la práctica lo mejor posible.

Por este motivo, y aunque según las bases de la práctica no cuenta en la nota, no hemos dejado de lado tampoco la estética del proyecto. Aunque poco, hemos dedicado tiempo a la selección de personajes y la inclusión de elementos visuales bonitos en la escena, como puede ser una base petrolera con su correspondiente animación en la escena general o una explosión de los agentes arqueros. Consideramos que merece la pena hacer la práctica visualmente más agradable tras el esfuerzo dedicado en cada parte de la misma.

Para concluir esta documentación hemos anotado de forma aproximada el tiempo dedicado a cada parte, esperando que pueda mejorar la asignatura en el futuro.



Steerings básicos	20h
Steerings delegados	30h
Formación	80h
PathFinding	100h
WallAvoidance	10h
PathFollowing	10h
Árbitros	7h
Documentación	15h
Total	272h

Bloque 2	
Sistema de combate	2h
Escenario	6h
Condiciones de victoria	1h
Árboles de decisión	10h
Mapa de influencia	8h
Guerra Total	1h
Opcionales	2h
Documentación	6h
Total	36h



De igual forma, nos gustaría mostrar una imagen del repositorio del proyecto, el cual cuenta con más de 240 commits como vemos en Figura 31 y donde su desarrollo ha sido constante entre los meses de la asignatura como mostramos en Figura 32.

**About**  
No description, website, or topics provided.  
[Readme](#)

**Releases**  
No releases published  
[Create a new release](#)

**Packages**  
No packages published  
[Publish your first package](#)

**Contributors** 3  
**MonsignorEduardo** Eduardo  
**Javixuwe** TheJavixuWe  
**thetigran** Javier Nicolas

**Languages**

**Code** **Issues** **Pull requests** **Actions** **Projects** 1 **Wiki** **Security** **Insights**

This branch is 19 commits ahead, 3 commits behind main.

Go to file Add file ▾ **Code** ▾

**MonsignorEduardo** Merge remote-tracking branch 'origin/Fixed' into Fixed e869850 15 minutes ago 243 commits

- vccode Auto stash before merge of "main" and "origin/main" 10 days ago
- Assets Merge remote-tracking branch 'origin/Fixed' into Fixed 15 minutes ago
- Packages Camara added 20 hours ago
- ProjectSettings Merge remote-tracking branch 'origin/main' into main 20 hours ago
- UIElementsSchema Arboles cambiados 2 days ago
- UserSettings Jamon 2 hours ago
- hooks Version usada 2020.3.7f1 2 months ago
- lts/objects/aa/a9 1 parte hecha empezamos con la segunda 3 days ago
- .gitattributes Initial commit 4 months ago
- .gitignore Initial commit 4 months ago
- .vsconfig Posición to Angulo 3 months ago
- Assembly-CSharp.csproj.DotSettings AttacMode 22 hours ago
- README.md Añadidos hats en resto de escenas 5 days ago

README.md

**Proyecto para IA PARA EL DESARROLLO DE JUEGOS**

Los autores conectados en implementar algunos elementos de inteligencia artificial. En un primer momento se

Figura 31. El repositorio en GitHub del proyecto cuenta con cerca de 250 commits en el momento de la imagen. Quién sabe si se superará en actualizaciones futuras.



Figura 32. Comenzamos la práctica en el repositorio el día 7 de marzo y desde ese periodo se ha actualizado de forma constante. Notamos una parada a finales de mayo al ser época de exámenes, y una subida en semanas previas a la entrega.

## 8. REFERENCIAS

- [1] Modelo del agenteNPC soldado: <https://github.com/Unity-Technologies/ml-agents>
- [2] Modelo del agenteNPC tanque:
- [3] <https://assetstore.unity.com/packages/essentials/tutorial-projects/tanks-tutorial-46209>
- [4] Modelo del agenteNPC arquero:  
<https://assetstore.unity.com/packages/3d/vehicles/land/free-surfer-mini-van-with-cartoon-style-option-154558>
- [5] Árboles: <https://github.com/yoshidan/UniBT>