Documentación Proyecto - Junio 2019

IA para el Desarrollo de Videojuegos

Grado en Ingeniería Informática

Norberto García Marín
Óscar García Martínez

Profesores: ***Luis Daniel Hernández Molinero***
Francisco Javier Marín-Blázquez Gómez

Índice

1. Introducción

Esta práctica tiene como objetivo la implementación de un sistema de inteligencia artificial enfocada a los videojuegos. Está dividida en dos bloques. En el primer bloque se implementarán los elementos asociados al movimiento y en el segundo bloque se implementarán los elementos necesarios para el desarrollo de una IA táctica y estratégica en un juego de guerra a tiempo real.

2. Software Utilizado

El software utilizado para la realización de esta práctica ha sido el motor de videojuegos multiplataforma Unity en su versión 2018.3.14f1, ya que ha sido la herramienta que hemos utilizado durante el primer cuatrimestre y estamos más familiarizados con ella.

3. Bloque 1

En este bloque detallaremos todas las clases que hemos creado para la realización de esta parte de la práctica.

Por un lado tenemos las clases utilizadas para implementar a los agentes y por otro las clases que implementan los steerings.

3.1. Estructuras principales

3.1.1. Kinematic

Esta será la clase principal, de la cual hereda **Agent**, y en ella se almacenan los atributos principales de cualquier objeto en un mapa, que son:

- Posición
- Orientación
- Velocidad
- Rotación

Cuenta con la implementación de los métodos necesarios para actualizar dichos atributos en función de los atributos devueltos por los steerings, pero eso se verá más adelante.

También guarda otros atributos como la máxima velocidad, aceleración, rotación y aceleración angular que puede tener un objeto.

3.1.2. Rol

La clase **Rol** es una clase abstracta de cual heredan los distintos roles implementados (**Ligero**, **Patrullador** y **Pesado**) y tiene dos atributos, *Velocidad* y *Aceleración*, los cuales indican la velocidad y aceleración máxima de un agente y su valor dependerá del rol que se elija.

3.1.3. Agent

La clase **Agent** hereda de la clase **Kinematic**. Esta clase contiene el rol que va a tener dicho agente y establece sus valores máximos en función de dicho rol.

También proporciona una función para modificar esos valores, aparte de por su rol, por el tipo de terreno en el que se encuentre y es la encargada de pintar los Gizmos si así se indica.

3.1.4. AgentNPC

Esta clase es una clase abstracta que hereda de **Agent** e implementa la interfaz **SteeringApplier**. Contiene una lista de Steerings y declara la función abstracta *applySteering*, que recibe como parámetro un **Steering** (en la siguiente sección se verá qué es). A partir de esta función se pueden crear distintos tipos de agentes, los cuales podrán implementar de una manera u otra la forma en la que se aplican los steerings.

3.2. Steerigns

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.

Para la implementación de los steerings hay dos clases principales:

- **Steering**: Una clase con dos atributos (*lineal* y *angular*) que indican la velocidad lineal y angular con la que se tendrá que desplazar el personaje.
- **SteeringBehaviour**: Esta es la clase abstracta base para cualquier steering que se vaya a implementar. Tiene un atributo de tipo **Kinematic** que indica el objetivo/target del steering y la función abstracta *getSteering*, que recibe un **Agent** (correspondiente al agente que llama a dicho steering) y devuelve un **Steering**. Esta función será la encargada de calcular el desplazamiento que debe realizar un agente en función al tipo de steering que la implemente.

Podemos diferenciar varios tipos de steerings:

- Básicos: Seek, Flee, Arrive y Wander.

- Acelerados: Seek Acelerado, Flee Acelerado, Arrive Acelerado, Align y Velocity Matching.
- Delegados: Pursue, Evade, Face, Looking WhereYouGoing, Wander, Path Following, Collision Avoidance y Wall Avoidance.
- De grupo: Separation, Alignment y Cohesion.
- Compuestos: Flocking, Leader Following y Blended Steering.
- Coordinados: Movimiento en formación.

3.2.1. Steerings Acelerados

3.2.1.1. Seek Este steering intenta hacer coincidir la posición del personaje con la posición del objetivo que se le indique.

Para ello se le indica el objetivo al que quiere llegar, se calcula la dirección del vector restando la posición del objetivo menos la posición del personaje, se normaliza y se le aplica una aceleración (la aceleración angular se mantiene a 0), hasta alcanzar su objetivo.

3.2.1.2. Flee Al contrario que con el **Seek**, este steering busca huir de un objetivo. El procedimiento es igual que el anterior, salvo que esta vez se resta la posición del personaje menos la posición del objetivo.

3.2.1.3. Arrive Este steering es parecido al **Seek** ya que su propósito es llegar a la posición del objetivo, pero en este caso la velocidad varía a lo largo del recorrido.

Para ello, el objetivo cuenta con dos radios, un radio interno y otro externo. Si el personaje se encuentra fuera del radio externo del objetivo, este se desplazará a máxima velocidad. Una vez entra dentro de dicho radio, su velocidad se va reduciendo hasta que pasa el radio interno y termina por pararse.

3.2.1.4. Align Con este steering lo que se consigue es que el personaje mire en la misma dirección que su objetivo, osea que tenga su misma orientación.

El funcionamiento es similar al del **Arrive**. Se cuenta con dos radios, uno para indicar lo que queda para tener la misma orientación que el objetivo y otro a partir del cual se reduciría la velocidad de rotación. En este caso, en vez de tratar de velocidades y posiciones se manipulan rotaciones y orientaciones. Se ha implementado la función *MapToRange*, la acota la rotación que hay que hacer entre $-\pi$ y π. Si dicha rotación es mayor que el ángulo exterior se le asigna la máxima rotación y una vez se pasa dicho ángulo se va decrementando la rotación hasta pararse.

3.2.2. Steerings Delegados

Estos tipos de steerings se basan en los steerings anteriores y añaden cierta funcionalidad.

3.2.2.1. Pursue Aquí nos encontramos con lo que sería una persecución, por lo que este steering hereda de **Seek**. No es un seguimiento simple como sería el del Seek, sino que aquí se intenta predecir, en función de la velocidad del personaje, hacia dónde se mueve el objetivo.

Primero se obtiene la distancia entre el objetivo y el personaje. Si la velocidad del personaje es menor que la distancia entre una máxima predicción que establecemos nosotros se establece la predicción como esa máxima, si no, la predicción será la distancia entre la velocidad del personaje. Por último se actualiza la posición a la que se tiene que desplazar el personaje con la velocidad del objetivo y la predicción calculada y se realiza un Seek a dicha posición.

3.2.2.2. Evade Este steering funciona de manera similar al **Pursue** salvo porque hereda de **Flee**, por lo que huirá del objetivo.

3.2.2.3. Face El objetivo de este steering es que el personaje miré en la dirección donde se encuentre su objetivo y hereda de **Align**.

Para ello, se calcula la orientación en la que se encuentra el objetivo y se hace un Align a esa orientación.

3.2.2.4. Looking WhereYouGoing Este steering hereda de **Align** y con él se consigue que un personaje mire hacia donde se está dirigiendo su objetivo. La orientación del objetivo se calcula en función de la velocidad del personaje.

3.2.2.5. Wander Este steering hereda de **Face** y tiene un comportamiento un tanto aleatorio, pues el personaje deambula por el terreno sin rumbo fijo y variando su dirección constantemente.

En primer lugar se crea un objetivo ficticio y se le asigna la orientación que tiene el personaje mas un número random entre -5 y 5 multiplicado por la variable *wanderRate*. Después se le asigna la posición del personaje mas la variable *wanderOffset* por la orientación en Vector del personaje. Por último, se hace un Face a la nueva posición y se aplica una máxima aceleración lineal en la nueva orientación.

3.2.2.6. Path Following Este steering hereda de **Seek** y permite a un personaje seguir un camino (path) determinado. Al inicio se crea un objeto de la clase Path, que contiene una lista con los nodos del camino, después se le asigna como objetivo al personaje el primer nodo de dicho Path y se hace un Seek a esa posición. Cada uno de los nodos que forman

el Path es un objeto de la clase Nodo, los cuales guardan su posición dentro del Path, su posición en el plano y un radio de proximidad. Cuando el personaje sobrepasa dicho radio, se le asigna el siguiente nodo como objetivo.

Se ha añadido la opción de elegir si queremos que cuando llegue al último nodo vaya de nuevo al primero o se de la vuelta y haga el camino a la inversa.

3.2.2.7. Wall Avoidance Este steering hereda de **Seek** y tiene como objetivo evitar que un personaje choque contra objetos estáticos, en nuestro caso cualquier objeto con el layer "*Muro*". Para ello tenemos tres clases.

La clase *Collision* guarda la posición donde se detecta una colisión y el vector normal en el punto de colisión. La clase *CollisionDetector* tiene la función *GetCollision* que comprueba dada una posición de origen, una dirección y una distancia máxima si hay algún objeto con el layer "*Muro*". Si detecta algún objeto devuelve un objeto de tipo Collision con la posición de dicho objeto y su normal. Por último, la clase *WallAvoidanceSteering* que implementa el steering en si. Le añade al personaje tres "bigotes"que funcionan como sensores, uno en la dirección en la que se mueve el personaje y otros dos a 30º y -30º respecto al primero y con un tamaño menor que este. Por cada sensor, se llama a la función **GetCollision** con la posición del jugador, la dirección del sensor y la distancia de este y si devuelve una colisión, se hace un Seek a la normal de dicha colisión.

3.2.3. Steerigns de Grupo

Estos steerings están pensados para utilizarlos con grupos de personajes.

3.2.3.1. Separation Este steering nos permite que un grupo de personajes se mantengan a una determinada distancia entre ellos. Para ello, cada personaje recorre una lista con el resto de personajes que formarán dicho grupo y calcula la aceleración linear en la dirección necesaria para que se separe de los personajes cercanos.

3.2.3.2. Alignment Este steering hace que un grupo de personajes vayan orientados en la misma dirección.

3.2.3.3. Cohesion Este steering dirige a un personaje al centro de masas de un grupo de personajes. Para calcular el centro de masas se suman las posiciones de los personajes que hay dentro de un radio determinado y se divide entre el total de esos personajes. Después se hace un **Seek** a ese centro.

3.2.4. Steerings Compuestos

Los steerings compuestos son steerings que se crean con la utilización de varios steerings a la vez.

3.2.4.1. Blended Steering Este steering declara una clase llamada **BehaivourAndWeight**, que se compone de un steering y un peso. Blended Steering tiene una lista de estos *BehaivourAndWeight* y para calcular el movimiento del jugador recorre dicha lista y acumula el valor de los steerings multiplicados por el peso. Finalmente se devuelve el steering acumulado al jugador.

Los siguientes steerings heredan de este y lo único que harán será añadir los steerings que quieran combinar a dicha lista.

3.2.4.2. Flocking Este steering espera un comportamiento como una bandada en grupo, guardando distancia entre ellos. Para ello se han añadido a la lista los steerings **Separation**, **Cohesion**, **Alignment** y **Wander** en ese orden.

3.2.4.3. Leader Following Como su nombre indica, este steering lo que hace es que un grupo de personajes sigan a un líder, manteniendo siempre una distancia dada. Para ello se han añadido a la lista los steerings **Arrive**, **Separation** y **Evade** en ese orden.

Para calcular el objetivo al que tiene que llegar el arrive, cogemos la velocidad del objetivo a seguir, se cambia su sentido, se normaliza y se multiplica por la distancia a la que queremos que estén.

3.2.5. Formaciones

Para la implementación de las formaciones hemos creado las siguientes clases:

3.2.5.1. SlotAssignment Representa una posición dentro de una formación y guarda el personaje al que se la asigna, la posición dentro de la formación y la posición que tiene dicho slot en el mundo real.

3.2.5.2. FormationPattern Es una clase abstracta que guarda quién es el líder de la formación (personaje a partir del cual se van a calcular el resto de las posiciones), una lista de Kinematics que representan las posiciones dentro de la formación, el número de slots que necesita, el máximo número de slots que soporta (por si se quiere hacer escalable) y la distancia al líder. Tiene también la función *GetSlotLocation* que recibe el número de la posición dentro de la formación y devolverá la posición del mundo real a la que se corresponde.

3.2.5.3. SquadPattern Define las posiciones para una formación en cuadrado y hereda de **FormationPattern**. El resultado es la formación que se ve en la imagen 2.

3.2.5.4. RupiaPattern Define las posiciones para una formación en forma de rupia (del Zelda) y hereda de **FormationPattern**. El resultado es la formación que se ve en la imagen 3.

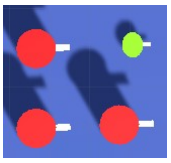

Figura 2: Formación en Cuadrado

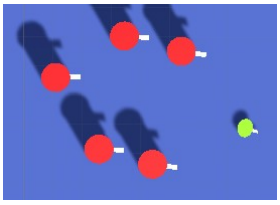

Figura 3: Formación de Rupia

3.2.5.5. FormationManager Esta clase guarda el líder de la formación, la lista de personajes que formarán la formación (menos el lider), una lista de **SlotAssignment** y la formación que se va a realizar.

En primer lugar, le asigna a cada persona de la lista un slot (añadiendo un SlotAssignment a la lista). En cada *Update* recalcula la posición de cada slot llamando a la función *GetSlotLocation* de la formación que tiene. Por último, tiene la función *GetSlotByCharacter* que dado un personaje devuelve la posición del slot al que pertenece.

3.2.5.6. Formation Steering Este es el steering que se le aplicará a cada miembro de la formación, menos al líder. Está compuesto de un **Arrive**, **Separation** y **Wall Avoidance** y tiene como atributo el **FormationManager** al que pertenece el personaje. Su funcionamiento es sencillo, le pide al FormationManager cuál es la posición en el mundo que ocupa su posición en la formación mediante la función *GetSlotByCharacter* y pone como objetivo del steering dicha posición.

3.2.6. Árbitro

La clase árbitro se encarga de añadir steerings ponderados a un agente atraves del steering del tipo **blended**. Gracias al uso de un controlador llamado **InpuntHandler** es capaz de obtener steerings ponderados según el rol del agente. Es decir, por cada rol hay un arbitro diferente.

Además, es posible añadir steerings de manera manual, en tiempo de ejecución, dejando de lado el rol de agente.

3.2.7. Pathfinding LRTA*

Para obtener el camino entre dos puntos se hace uso del **LRTA***. La diferencia en entre este algoritmo y el A*[1] es que en cada iteración este obtiene el nodo adyacente con menor coste y devuelve el camino conseguido hasta el momento, con lo cual el camino es obtenido en tiempo real.

Una clara diferencia es que este algoritmo no consigue un camino óptimo ya que el nodo escogido en cada iteración forma parte de la solución final.

Para la implementación de los algortimos de busqueda se hace uso de una clase **Grid** que contine una matriz formada con nodos utilizada para el correcto uso de cualquier algoritmo de pathfinding.

A su vez este algoritmo puede hacer uso de las heurísticas [2] **Manhattan, Chebychev y Euclidea**.

3.3. Escenarios de Pruebas

3.3.1. Escena 1

En esta escena podemos diferenciar dos bloques de personajes.

En el primero tenemos un personaje que controlamos nosotros y después 5 agentes que tienen distintos steerings, pero un objetivo común, el personaje controlado. Los steerings que tienen cada uno, respectivamente, son Seek, Flee, Arrive, Align y Evade.

El segundo bloque de personajes está compuesto por un personaje con un Wander y un grupo de personajes con un Following Leader

Figura 4: Escena 1

3.3.2. Escena 2

En este escena tenemos tres personajes, cada uno con un rol distinto y todos con su propio agente. Tenemos un personaje Ligero, uno Pesado y un Patrullador. Por defecto, el ligero y el patrullador ejecutarán un Wander, el pesado usará un Pursue sobre el personaje ligero y todos tienen un WallAvoidance.

Figura 5: Escena 2

3.3.3. Escena 3

En esta escena tenemos dos formaciones, una formación con forma de Rupia cuyo líder tiene un Wander y otra formación con forma de Cuadrado cuyo líder tiene un PathFollowing por los puntos del mapa. Por último tenemos un personaje en el centro del camino con un Face al líder de la formación cuadrada.

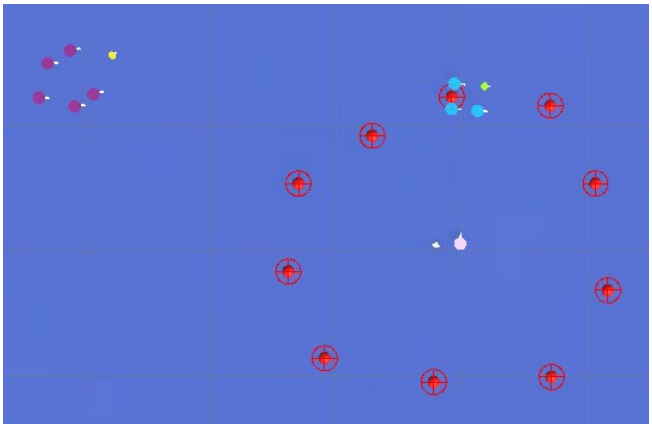

Figura 6: Escena 3

3.3.4. Escena 4

En esta escena hay tres personajes. Si pulsamos con el teclado el 1 seleccionamos al primer personaje, si pulsamos el 2 se seleccionara el segundo personaje y si pulsamos el 3 seleccionaremos al tercer personaje. Para quitar cualquier selección se debe pulsar el espacio.

Una vez se ha seleccionado un personaje, podemos seleccionar con el click izquierdo del ratón sobre el mapa un punto y el personaje seleccionado realizará un pathfinding hacia dicha posición.

Figura 7: Escena 4

3.3.5. Escena 5

En esta última escena lo que hay es un gran número de personajes con el comportamiento Flocking.

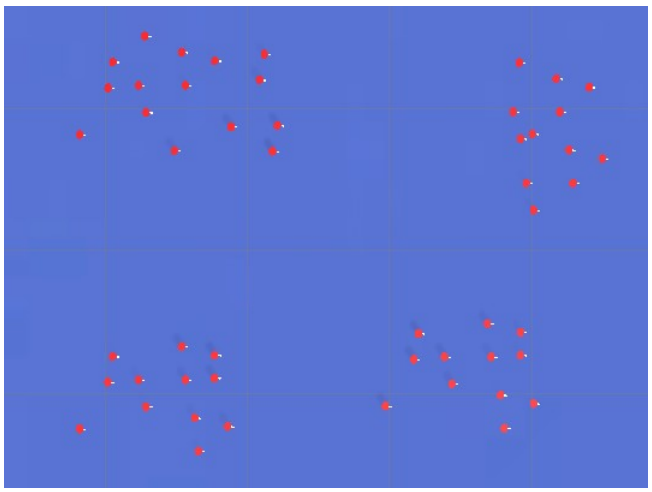

Figura 8: Escena 5

4. Bloque 2

4.1. Introducción

En este segundo bloque se explica la implementación de algunos elementos de inteligencia artificial en un juego de guerra. Como escenario para dicho juego tenemos un mapa cuadrado con diferentes zonas (explicadas más adelante) diferenciadas por sus colores, y una serie de agentes, diferenciados por sus formas y tamaños, con diferentes roles que les harán tener un comportamiento u otro a lo largo del juego. El objetivo final sería que los agentes de un bando llegasen a la base del otro y que un agente se mantenga ahí durante 5 segundos.

Para el desplazamiento de los agentes por el mapa se utilizará el algoritmo A*[1], el cuál es un algoritmo heurístico que dada una posición inicial y una posición final, busca el camino más corto estudiando paso a paso las posiciones adyacentes a las que se encuentra el agente y, en función de su heurística, escoger la posición de menor coste que le llevará a la posición final. Dicha heurística viene dada por el peso que tiene cada tipo de terreno y por el mapa de influencias que alimentan los personajes.

4.2. Mapa

Nuestro mapa está dividido en las siguientes zonas:

- Césped - Pintadas de verde.
- Calzadas - Pintadas de gris.
- Agua - Pintadas de azul claro.
- Puentes - Pintadas de marrón.
- ZonasSeguras - Pintadas de rosa claro - Si un agente pasa por estas zonas y su salud no está al máximo, esta incrementará.
- Veneno - Pintadas de violeta - Si un agente pasa por estas zonas su salud se irá reduciendo hasta que salgan de ella.
- BaseRoja - Pintada de rojo.
- Base Negra - Pintada de **negro**.

Aparte de estos elementos visibles, en cada lado del mapa también hay unos puntos denominados waypoints que serán objetivos de ataque/defensa en función del comportamiento que tengan los agentes en cada momento. En la imagen 9 se muestran dos imágenes, una con el mapa y otra donde se muestran los waypoints.

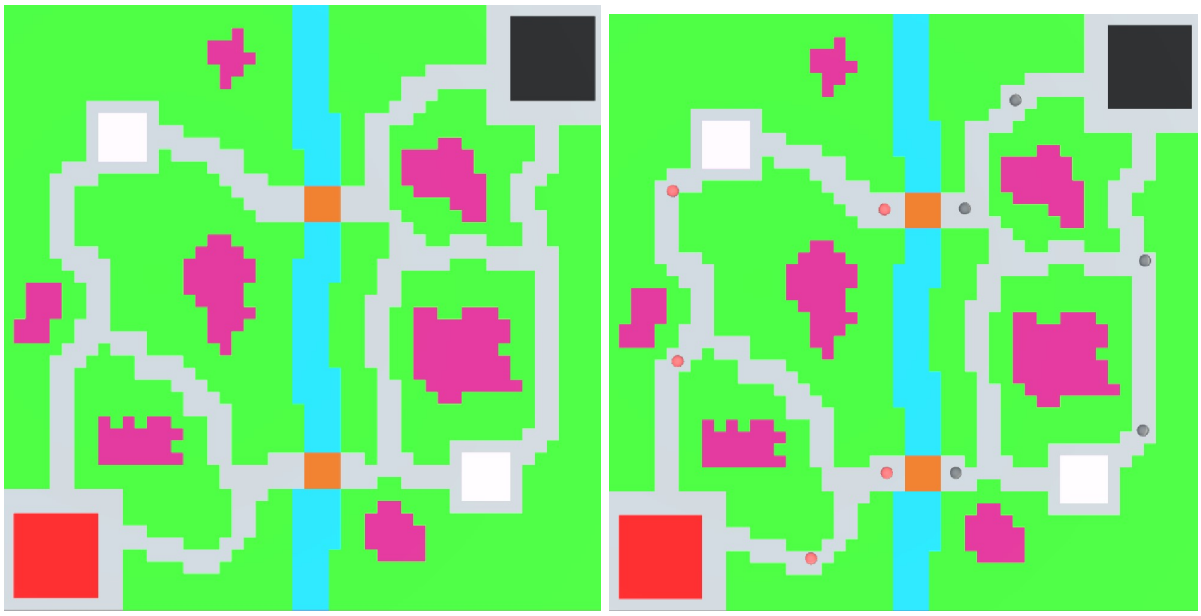

Figura 9: Mapa

4.3. Agentes

En función del rol que desempeña cada agente podemos distinguir los tipos que se ven en la tabla 1:

Rol/Atributos	Salud	Puntos de Ataque	Puntos de Defensa	Rango de Ataque
Ligero	200	30	10	1
Patrullador	250	35	7	1
Pesado	350	45	15	1
Arquero	200	25	5	4

Cuadro 1: Relación entre tipo de rol y sus atributos.

El rol indica al agente cuál es su salud inicial y máxima, sus puntos de ataque y sus puntos de defensa. Los tres primeros realizarán ataques de tipo melee (golpearán al enemigo si están a su lado) y el **Arquero** realizará ataques de tipo ranged (golpeará al enemigo si se encuentra dentro de un determinado rango de distancia).

Para calcular la fuerza con la que ataca un agente a un enemigo se usa la siguiente fórmula:

$$FAtaque = Ataque * FAD * FTA$$

El Factor de tipo de atacante/defensor (FAD) viene dado en función del rol que tiene el agente atacante y el rol del agente al que ataca, como se ve en la tabla 2, y el factor por terreno del atacante (FTA) depende del tipo de rol del atacante y del tipo de terreno sobre el que se encuentre, como viene dado en la tabla 3.

Se ha implementado la posibilidad de que se realice un ataque potente con una probabilidad de un 1 %. En ese caso se multiplicará el ataque del agente por una constante, propiciando así un golpe crítico.

$$FAtaque = Ataque * CteImpacto$$

Atacante/Defensor	Ligero	Patrullador	Pesado	Arquero
Ligero	x1	x1.25	x0.75	x1.30
Patrullador	x1.50	x1	x0.5	x1.25
Pesado	x1.25	x1.50	x1	x1.50
Arquero	x1.15	x0.95	x1.25	x1

Cuadro 2: Factor de tipo de atacante/defensor (FAD)

Unidad/Terreno	Calzada/Puente	Cesped	Veneno
Ligero	x1.25	x1.10	x0.85
Patrullador	x1.05	x0.90	x1
Pesado	x1.15	x1	x1.05
Arquero	x1	x1.15	x1.05

Cuadro 3: Factor por terreno del atacante (FTA)

La fuerza con la que un agente se defiende depende de sus puntos de defensa (que vienen indicados en su rol) y el tipo del terreno en el que se encuentre, calculándose con la siguiente fórmula:

$$FDefensa = Defensa * FTD$$

De igual forma que con el ataque, existe un 1 % de posibilidades de que se realice una defensa máxima, en la cual se multiplicará su poder de defensa por una constante:

$$FDefensa = Defensa * CteDefensa$$

Unidad/Terreno	Calzada/Puente	Cesped	Veneno
Ligero	x1	x0.95	x0.75
Patrullador	x1	x1.15	x0.95
Pesado	x1	x0.95	x1.05
Arquero	x1	x0.95	x0.90

Cuadro 4: Factor por terreno del defensor (FTD)

4.3.1. Ligero

Los agentes ligeros tienen dos comportamientos. El comportamiento por defecto o defensivo es elegir el waypoint de su lado del mapa más cercano que tenga menos agentes defendiéndolo y hacer guardia en él a la espera de que lleguen enemigos. Si se pone en modo ofensivo su objetivo será ir avanzando por los waypoints más cercanos del bando contrario hasta llegar a la base enemiga.

4.3.2. Patrullador

Este personaje tiene como objetivo defender su territorio. Mientras ningún enemigo se acerque por el territorio estarán patrullando cerca de su base, pero si un enemigo llega a uno de los waypoints de su territorio, el patrullador se irá al waypoint que haya después del atacado como refuerzo por si el enemigo consigue sobrepasar el waypoint en el que se encuentra. Si un enemigo llega a su base, dejará lo que esté haciendo y se irá a por él.

4.3.3. Pesado

Los agentes pesados tienen como objetivo ir a por la base enemiga. Para ello seleccionarán el waypoint enemigo más cercano y se dirigirá a él. Una vez logré alcanzarlo y elimine a los enemigos, si los hay, se dirigirá al siguiente, así hasta llegar a la base enemiga.

4.3.4. Arquero

Los arqueros tienen el mismo comportamiento que los agentes ligeros, salvo porque son capaces de atacar desde una determinada distancia.

4.4. Diagrama IA táctica

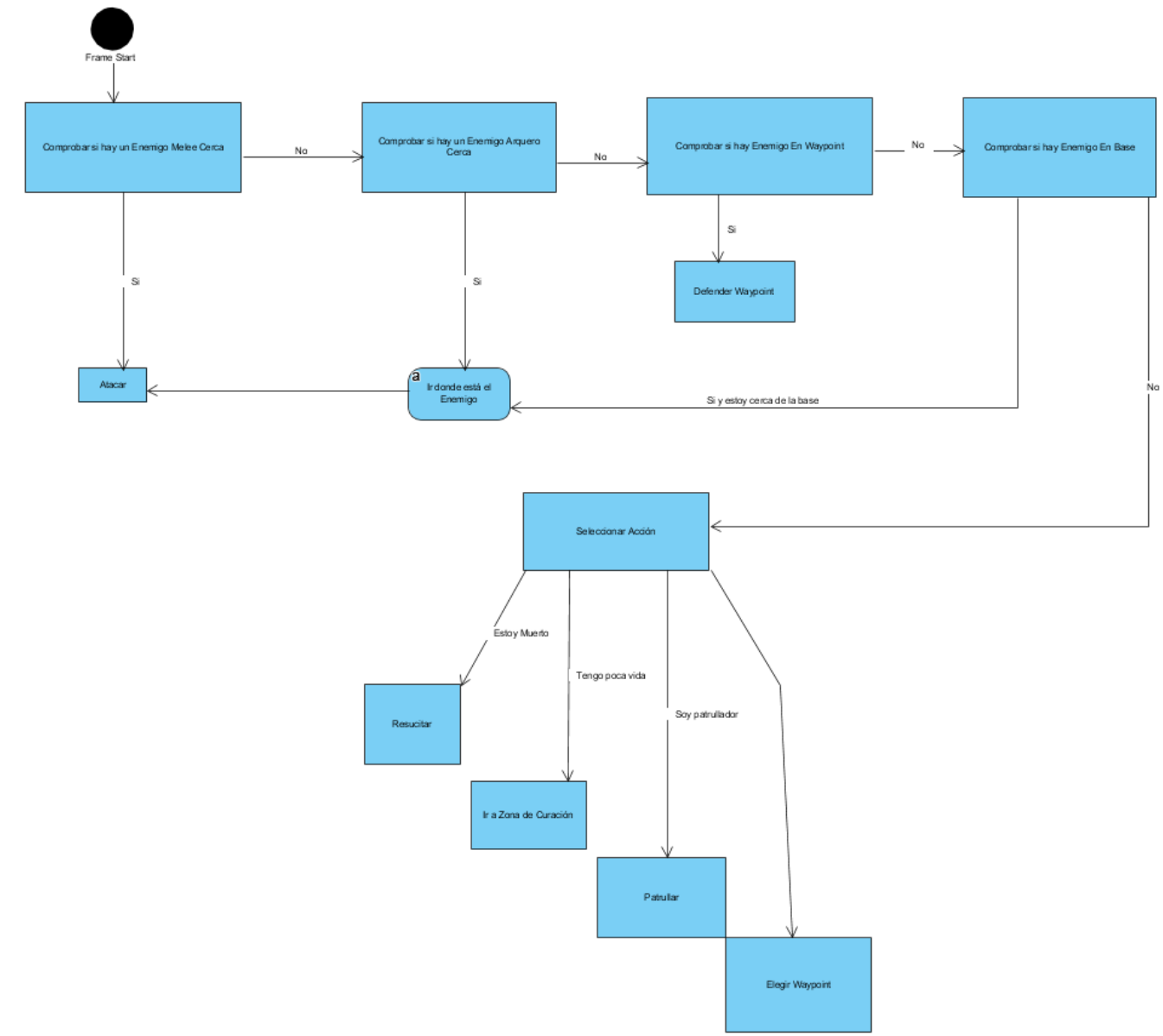

Figura 10: Diagrama

Lo primero que hace la IA es comprobar si un enemigo está en su rango de ataque con lo cual tienen que atacarse incondicionalmente. Si no ocurre esto se comprueba si hay algún enemigo arquero, y si lo hubiese tendría que ir a por él (si no esta en su rango).

Si nada de esto ocurre se comprueba si hay enemigos en un Waypoint y si es así se dirige al Waypoint. Si no hay enemigos comprueba si hay algún enemigo en base y si le concierne a él dirigirse a la base a defender.

Si no es así se debe seleccionar una acción. Primero, se comprueba si está muerto y tendría que resucitar, si no se comprueba si tiene poca vida y si es así tendría que ir a la

zona de curación. Si no se cumple nada de lo anterior se comprueba si su rol es patrullador, con lo cual deberá patrullar al rededor de la base.

Si nada de lo anterior ocurre se debe elegir un Waypoint al que ir dependiendo de si se esta en modo ofensivo, defensivo o guerra.

4.5. Interfaz

En la imagen 11 se muestra la interfaz de la aplicación.

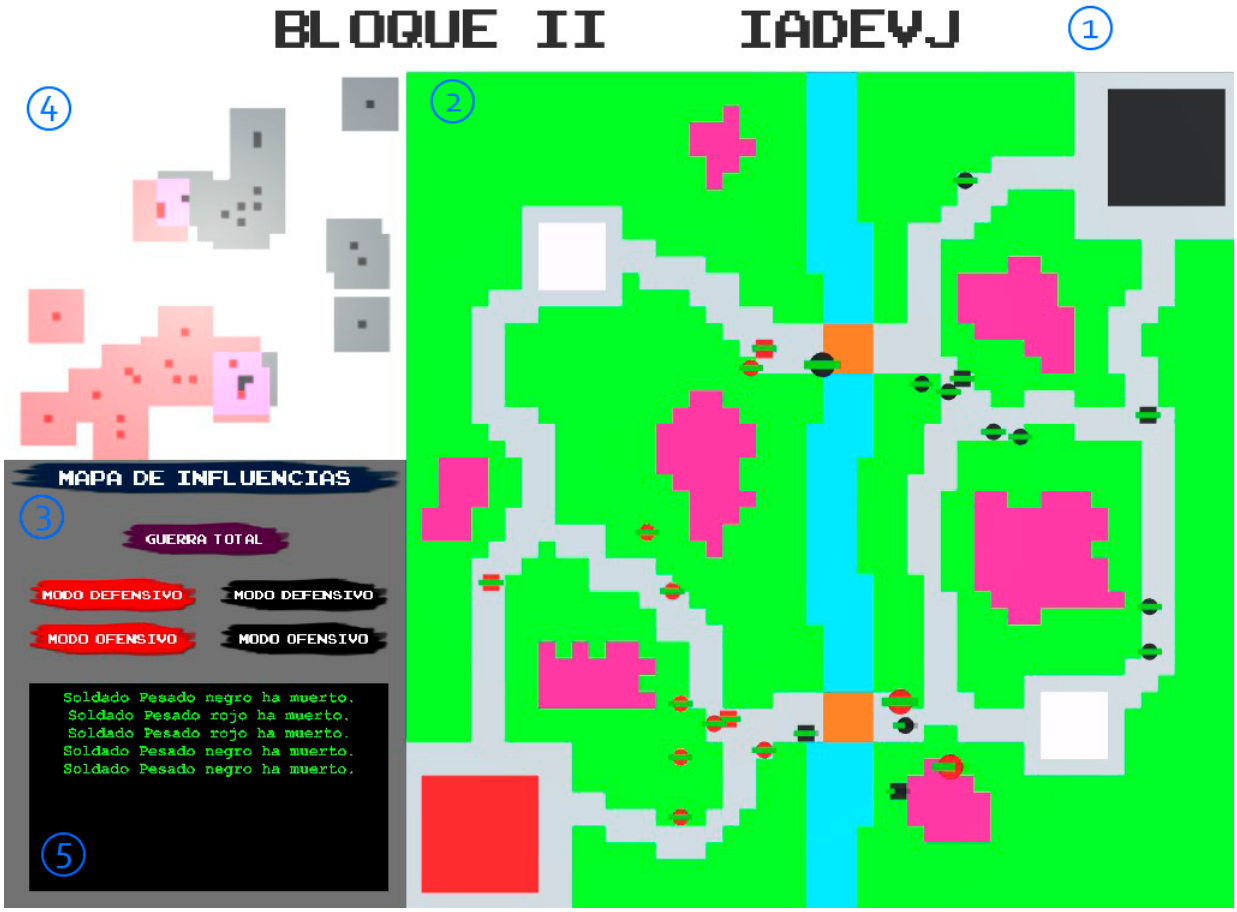

Figura 11: Interfaz

Esta interfaz está compuesta por 5 partes:

1. Título de la aplicación.
2. Mapa del juego.
3. Menú con botones.
4. Mapa de influencias de los personajes del juego.
5. Consola de texto donde se mostrará información sobre la partida.

Los usuarios pueden interactuar tanto con los botones como con la zona del mapa.

4.5.1. Interacción con Botones

Se pueden distinguir 3 botones:

4.5.1.1. Botón Modo Defensivo Cuando se pulsa este botón se cambia el modo de los agentes ligeros y los arqueros del equipo correspondiente a modo defensivo. En el caso de haber seleccionado un agente del color pulsado, se le pondrá a él dicho modo.

4.5.1.2. Botón Modo Ofensivo Cuando se pulsa este botón se cambia el modo de los agentes ligeros y los arqueros del equipo correspondiente a modo ofensivo. En el caso de haber seleccionado un agente del color pulsado, se le pondrá a él dicho modo.

4.5.1.3. Botón Modo Ofensivo Cuando se pulsa este botón se cambia el modo de los agentes ligeros, los pesados y los arqueros de ambos equipos a modo guerra total e intentarán ir en todo momento a por la base enemiga.

4.5.2. Interacción con el Mapa

Existe la opción de poder seleccionar un agente y darle un objetivo. Para ello se seleccionará el agente deseado en el mapa haciendo click con el botón izquierdo del ratón sobre el agente que se desee mover. En ese momento el agente cambiará de color para destacar que ha sido seleccionado, como se ve en la imagen 12.

Después se selecciona el lugar al que lo quieres enviar haciendo click con el botón derecho del ratón sobre el mapa y si es una posición a la que pueda acceder (no es agua) realizará el desplazamiento hasta dicha posición.

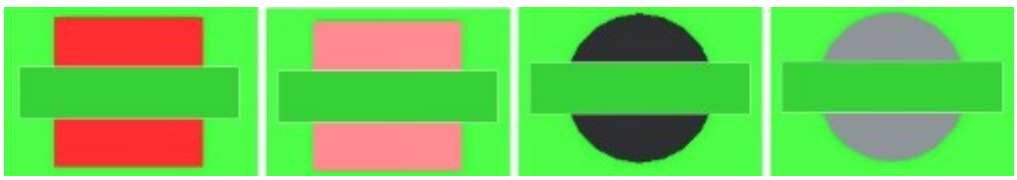

Figura 12: Agentes Libres/Seleccioandos

4.5.3. Dentro de Unity

Es posible ajustar variables (en tiempo de ejecucion o no) dentro del entorno de Unity. Es interesante ver como cambian los resultados.

4.5.3.1. Controlador Dentro del gameobject **Controlador** se es capaz de modificar el tiempo de resurrección, un factor de velocidad de movimiento y el tiempo necesario para la victoria de un equipo. Todo esto se encuentra respectivamente en las variables **Resucitar Time**, **Move Speed Rate** y **Victory Time**.

4.5.3.2. Controlador Mapa Influecia Se puede cambiar el intervalo de actualización del mapa de influencia. La variable **Update Interval** contiene el intervalo de actualización para un personaje, es decir, cuando menor sea más rápido se intentará actualizar el mapa.

Además la variable **Radio** representa la cantidad de nodos adyacentes en linea recta que se calculan para cada personaje.

Referencias

[1] Patrick Lester, A* Pathfinding for Beginners, 18 Julio, 2015. https://web.archive.org/web/20170509000025/http://www.policyalmanac.org/games/aStarTutorial.htm

[2] Amit Patel, Heuristics. http://theory.stanford.edu/ amitp/GameProgramming/Heuristics.html