Universidad de Murcia

Facultad de Informática

IA para el Desarrollo de Videojuegos

Real Time Wargame

Autores

Antonio López Martínez-Carrasco

antonio.lopez31@um.es

José María Sánchez Salas

josemaria.sanchez12@um.es

Profesores

Francisco Javier Marín-Blázquez Gómez

jgmarin@um.es

Luis Daniel Hernández Molinero

ldaniel@um.es

Índice

I Introducción y estructura de la aplicación **4**

1 Introducción **4**

2 Estructura de la aplicación **4**

II Manual de uso **6**

3 Manual de uso **6**

III Clase principal, mapa e interacción con el usuario **8**

4 Clase principal **8**

5 Mapa **9**

6 Interacción con el usuario **10**

IV IA reactiva **11**

7 Steerings **11**
7.1 Interfaz Steering . . . 11
7.2 Steering Uniforme . . . 11
7.3 Steering Uniformemente Acelerado . . . 11

8 Comportamientos **11**
8.1 Comportamientos no acelerados . . . 12
8.2 Comportamientos acelerados . . . 12
8.3 Comportamientos delegados . . . 14
8.4 Comportamientos de grupo . . . 16

9 Árbitros **16**

10 Modelo **17**
10.1 Clase WorldObject . . . 17
10.2 Clase Character . . . 18
10.3 Clase Obstacle . . . 18
10.4 Formaciones . . . 19

V PathFinding **21**

11 PathFinding **21**
11.1 Distancias . . . 21
11.2 Algoritmo LRTA* . . . 22
11.3 Clase PathFinding . . . 22
11.3.1 Pathfinding continuo . . . 22
11.3.2 Pathfinding punto a punto . . . 23
11.4 Pathfinding táctico . . . 23

VI IA táctica **24**

12 Modificaciones en el modelo **24**
12.1 Clase Character . . . 24
12.2 Enumerado Team . . . 25

13 Otros comportamientos **25**
13.1 Comportamiento de ataque . . . 25
13.2 Comportamiento de cura . . . 25
13.3 Ataque y cura en las formaciones . . . 26

14 Acciones y comprobaciones **26**

15 Roles Tácticos **27**
15.1 Roles defensivos . . . 27
15.2 Roles ofensivos . . . 30

16 Waypoints **31**

17 Puntos de moral **33**

18 Mapas de influencia **34**

VII Flocking **36**

19 Flocking **36**

VIII Elementos opcionales **37**

20 Elementos opcionales **37**

IX Conclusiones **38**

21 Conclusiones **38**

X Bibliografía **39**

Parte I
Introducción y estructura de la aplicación

1 Introducción

Este documento corresponde con la documentación del proyecto Real Time Wargame propuesto por la asignatura IA para el Desarrollo de Videojuegos perteneciente a la mención de Computación del 4º curso del Grado de Ingeniería Informática impartida por la Universidad de Murcia.

El proyecto consiste en implementar algunos elementos de inteligencia artificial en un entorno de juego de guerra en tiempo real. Estos elementos de inteligencia artificial van desde la parte reactiva, como comportamientos más básicos, como puede ser moverse de un sitio a otro, hasta la parte táctica, como comportamientos tácticos basados en una máquina de estados.

Para la implementación de este proyecto, hemos decidido utilizar la librería gráfica LibGDX [1]. El motivo por el cuál nos hemos decantado por esta librería es porque es una librería sin licencia, implementada en Java, con muy buena documentación y que no requiere de un hardware específico para su ejecución, además de que estamos muy acostumbrados a trabajar en Java. Así pues, no hemos implementado este proyecto para ningún hardware específico, simplemente hemos hecho uso del IDE Eclipse junto con la librería LibGDX, programando y ejecutándolo en nuestros respectivos ordenadores.

Este documento consta de ocho partes fundamentales: la primera de ellas se encarga de introducir el proyecto, así como también la estructura que tiene el proyecto desarrollado en Java. La segunda contiene un breve manual de uso para los usuarios. La tercera contiene la explicación de la clase principal del programa, el mapa implementado y la interacción con el usuario. La cuarta parte comenta toda la parte de la inteligencia artificial reactiva que hemos implementado, así como el modelo de objetos del videojuego que hemos desarrollado. La quinta trata las diversas implementaciones del Pathfinding que hemos llevado a cabo. La sexta parte contiene la parte de la inteligencia artificial táctica que hemos implementado, así como las modificaciones que hemos llevado a cabo en el modelo mencionado en la sección cuarta para poder introducir la parte táctica al modelo. La séptima parte comenta la implementación que se ha llevado a cabo del Flocking pedido en el proyecto. Y por último, en la parte octava, se hace una recopilación de todos los elementos opcionales que hemos implementado.

Las dos últimas partes de este documento constan de las conclusiones que hemos obtenido tras la realización de este proyecto y la bibliografía consultada para el mismo.

2 Estructura de la aplicación

La estructura que hemos llevado a cabo en este proyecto ha sido una estructura en paquetes (típica estructura de cualquier proyecto en Java). Esta estructura es la siguiente:

- Primero se encuentra el paquete `com.mygdx.iadevproject` que contiene la clase principal del proyecto, así como la clase que crea todos los personajes del videojuego.
- Seguidamente se encuentra el paquete `com.mygdx.iadevproject.aiReactive` que contiene toda la parte reactiva de la inteligencia artificial: desde los comportamientos no acelerados, pasando por los comportamientos acelerados, delegados y de grupo; los árbitros y el pathfinding. Cada uno de estos elementos, se encuentra en su paquete correspondiente.
- El paquete `com.mygdx.iadevproject.aiTactical` contiene toda la parte táctica de la inteligencia artificial: los roles implementados, los estados de las máquinas de estados implementadas, así como los nodos del árbol de decisión implementado.
- El paquete `com.mygdx.iadevproject.model` contiene todo el modelo de objetos que hemos considerado para este proyecto.

- El paquete `com.mygdx.iadevproject.map` contiene todo lo referente a la creación del mapa, así como a partir de él, la inicialización de los grids necesarios para el pathfinding.
- El paquete `com.mygdx.iadevproject.mapOfInfluence` contiene la parte del cálculo de los mapas de influencia. Así como su visualización.
- El paquete `com.mygdx.iadevproject.userInteraction` encapsula la interacción del sistema con el usuario.
- Y por ultimo, el paquete `com.mygdx.iadevproject.waypoints` encapsula toda la creación y manejo de los waypoints del juego.

Para poder probar el correcto funcionamiento de todas las funcionalidades implementadas, se ha creado una carpeta de test que tiene la misma estructura que la carpeta de fuentes, y que contiene todos los test implementados de toda la funcionalidad del sistema: comportamientos, árbitros, pathfinding, roles, formaciones, etc.

Parte II
Manual de uso

3 Manual de uso

En esta sección se va a exponer un breve manual de uso para el usuario.

Existen dos maneras de ejecutar el programa:

- Haciendo doble click sobre el ejecutable (situado en `jar/juego.jar`).
- Ejecutando la orden `java -jar jar/juego.jar` en la terminal.

La diferencia fundamental entre las dos maneras es que con la primera, aquellas indicaciones que el programa vaya dando al usuario cuando este selecciona personajes, no aparecerán. En la segunda manera, estas indicaciones sí se mostrarán en la propia terminal.

Por defecto, al arrancar el programa, el modo en el que se inicia el juego es el modo de "batalla final"; esto es, ambos equipos intentan conseguir la victoria, además de evitar, también, la derrota. Así pues, se verá cómo los personajes defensivos tratan de defender su zona, mientras que los personajes ofensivos tratan de ir hacia la base enemiga.

Nuestro juego esta configurado inicialmente para el modo batalla final, aunque, se pueden crear y establecer otros muchos modos de juego solamente modificando el comportamiento táctico de los personajes (su rol). Con estas sencillas modificaciones en el código fuente (en la parte de creación de los personajes) se pueden conseguir otros muy distintos y variados sistemas de juego. A parte de todo esto, durante la propia partida se da cierta libertad al usuario para que pueda realizar ciertas acciones con los personajes. Para ello, tiene que realizar lo siguiente:

- **Seleccionar** el personaje (o los **personajes**) a los que quiera aplicar acciones. Para seleccionar un personaje simplemente se ha de clickar sobre él con el botón izquierdo del ratón. Si se quiere seleccionar varios personajes, se debe mantener pulsada la tecla `Control-Izquierdo`, de lo contrario, solamente se seleccionará uno. Es importante destacar que solamente se pueden seleccionar varios personajes si estos pertenecen al mismo equipo. Cuando un personaje es seleccionado, dejará de hacer lo que estaba haciendo para realizar todo lo que le mande el usuario.
- Para **liberar** a los **personajes** simplemente se ha de pulsar el botón derecho del ratón en cualquier lugar. Cuando se libera un personaje, este vuelve a tener su rol activado, por lo que hará lo que estaba haciendo antes de que el usuario lo seleccione.
- Tras haber seleccionado a los personajes, el programa mostrará por terminal, un listado de las posibles acciones que puede realizar. Simplemente se debe pulsar los números de aquellas acciones que se quieren realizar y seguir los pasos que indica el programa. Las posibles acciones que puede realizar el usuario son:
  - **Aplicar comportamientos accelerados**.
  - **Aplicar comportamientos delegados**.
  - **Aplicar un pathfinding**.
  - **Aplicar comportamientos en grupo**.
  - **Aplicar otros comportamientos**.
  - **Realizar una formación**.
- Es importante destacar que los comportamientos que se quieran hacer, se harán para todos los objetos seleccionados. Es decir, todos los objetos seleccionados realizarán la misma acción que ha mandado el usuario.
- También, hay que tener en cuenta que cuando un personaje es seleccionado por el usuario, puede ser atacado por sus enemigos y, en consecuencia, puede morir. Cuando un personaje que está seleccionado muere, al no tener su rol activado, este personaje seguirá estando seleccionado y haciendo las acciones que el usuario quiera (pero los enemigos no lo tendrán en cuenta a la hora de atacarle, porque está muerto). Esto también se aplica a las formaciones. Cuando se deselecciona un personaje que ha muerto, al activarse su rol, este verá que está muerto y hará las acciones correspondientes a esta situación.

- Una de las posibles acciones que puede realizar el usuario es crear formaciones. Para crear una formación es necesario que se seleccione, al menos, tres personajes. Además, tras crearse la formación esta se creará y no estará seleccionada por el usuario. Para seleccionarla, simplemente hay que clickar sobre cualquiera de los personajes de la formación; una vez seleccionada, se puede realizar con ella cualquiera de las otras acciones (incluso crear otra formación). Para eliminarla, es necesario que se seleccione y se deseleccione la formación.

El programa dispone de cierta funcionalidad que se puede activar/desactivar por el usuario. Esta funcionalidad es la siguiente:

- **Mover la cámara de posición**. Para ello se han de usar las flechas de dirección.
- **Alejar/Acercar la cámara**. Para alejar la cámara se utiliza la tecla `A`, mientras que para acercar, se utiliza la tecla `Q`.
- **Pausar/Reaunar el juego**. Para pausar el juego, se utiliza la tecla `P`, mientras que para reanudarlo, se utiliza la tecla `O`.
- **Mostrar/Ocultar mapa de influcencia** encima del mapa. Para mostrar el mapa de influencia, se utiliza la tecla `I`, mientras que para ocultarlo, se utiliza la tecla `U`.
- **Deshabilitar/Habilitar que haya ganador**. Por defecto, siempre hay un ganador, sin embargo, puede ser que el usuario quiera que no lo haya, por lo que para deshabilitarlo, puede utilizar la tecla `X`, mientras que para volver a habilitarlo, se utiliza la tecla `Z`.
- **Mostrar/Ocultar debug de los comportamientos**. Debido a que puede ser interesante visualizar los comportamientos de los personajes en el mapa, esta opción está habilitada por defecto, para ocutarlo se utiliza la tecla `N`, mientras que para mostrarlo otra vez, se utiliza la tecla `M`.

Parte III
Clase principal, mapa e interacción con el usuario

4 Clase principal

Todo lo referente a esta sección, se encuentra dentro del paquete `com.mygdx.iadevproject` del proyecto. La clase principal de nuestro proyecto es la clase `IADeVProject` y es la que contiene los siguientes elementos:

- **Constantes**. Estas son: el tamaño del mapa (alto y ancho); el tamaño de las celdas de los grids, así como el tamaño de estos (alto y ancho); valores de infinito, coste y terreno por defecto; y el tamaño de los objetos del mundo (alto y ancho, obtenidos a partir de la información del mapa).
- **Mapas**. Estos son: el mapa real que se dibuja; el mapa de costes para el PathFinding; el mapa de terrenos para modificar la velocidad de un personaje dependiendo del terreno por donde se mueve (exceptuando el mapa real, todos los demás son los que se denominan grids y todos tienen el mismo tamaño).
- **Variables globales**. Estas son: los objetos y obstáculos del mundo; el conjunto de objetos seleccionados por el usuario (utilizando el ratón); la cámara; el flag que indica si se dibujan, o no, las líneas de depuración de los distintos comportamientos; así como las variables necesarias para que desde los comportamientos se puedan dibujar las líneas de depuración. También están las bases y manantiales de los equipos, el controlador de la interacción con el usuario, las variables que indican si se pausa (o no) el juego y si se dibuja (o no) el mapa de influencia encima del mapa real, así como las variables que indican qué equipo ha ganado (si lo hay) y la variable que indica si puede haber (o no) ganador en la partida.

Esta clase se encarga de crear e inicializar todos los elementos anteriores, de renderizar el seguimiento del juego y, una vez terminado, eliminar todos los elementos creados para el renderizado. Para la creación de todos los personajes, se ha creado la clase `CreateCharacters` que proporciona un único método estático `createCharacters()`. También se encarga de inicializar los waypoints del juego, así como los mapas de influencia. Para esto, hace uso de los métodos estáticos proporcionados por las clases `Waypoints` y `SimpleMapOfInfluence` (véase secciones 16 y 18).

El control de la interacción del usuario con el juego se ha implementado en la clase `InputProcessorIADeVProject` (ver sección 6), que contiene todos los manejadores para las posibles acciones que puede realizar el usuario con el teclado y el ratón. Debido a que el usuario puede seleccionar objetos del mundo, cuando se realiza la selección para que añadan los objetos al conjunto de objetos seleccionados, se hace uso del método `addToSelectedObjectsList()` de la clase `IADeVProject`. Como también se proporciona la funcionalidad de eliminar la selección de objetos, dicha clase también proporciona el método `clearSelectedObjectsList()`.

Debido a que el tamaño de los distintos grids y el tamaño del mapa pueden no ser iguales (pues depende del tamaño de las celdas), es necesario hacer una conversión entre las posiciones reales del mapa y las posiciones correspondientes del grid. Por eso mismo, la clase `IADeVProject` proporciona los dos siguientes métodos:

- `mapPositionTOgridPosition() y gridPositionTOmapPosition()`. Ambos reciben como parámetros el tamaño de la celda del grid y un vector con la posición que se quiere convertir. El primero transforma una posición del mapa a una posición del grid (índices de la matriz que representa al grid), y el segundo hace la inversa: transforma una posición del grid a una posición del mapa.

También proporciona métodos para obtener las bases y manantiales de cada equipo, así como la posición de ambas.

5 Mapa

Todo lo referente a esta sección, se encuentra dentro del paquete `com.mygdx.iadevproject.map` del proyecto. El mapa que se ha diseñado para este proyecto se muestra en la Figura 5.

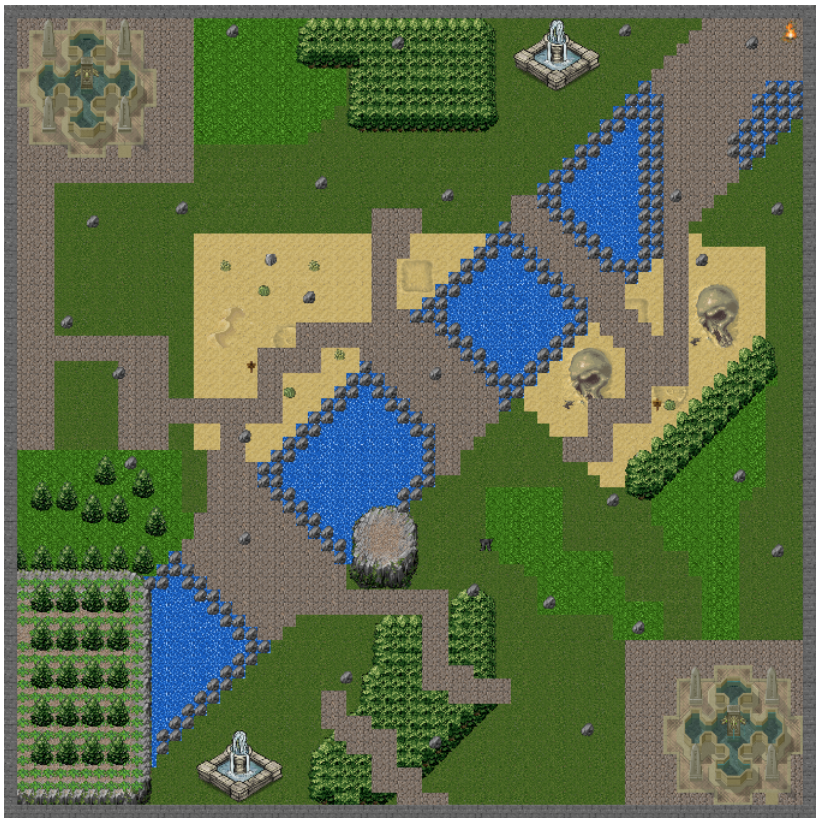

Figura 1: Mapa diseñado para el proyecto.

Como se puede observar, existen seis tipos de terrenos:

- **Agua y montañas**. Representan los terrenos infranqueables. Como indica el enunciado del proyecto, los dos países se encuentran separados por un río con cuatro puentes para cruzarlo.
- **Desierto y bosque**. Se identifican de manera directa en el mapa. También se tiene como bosque los árboles que se encuentran dentro la pradera situada en la parte inferior izquierda del mapa.
- **Pradera**. Se corresponde con los tramos de color verde más claro.
- **Sendero**. Se corresponde con el fondo del mapa.
- **Camino**. Se corresponde con los tramos de color gris. Las bases de los países se encuentran encima de este tipo de terreno.

Para representar los terrenos en el proyecto, se ha implementado el enumerado `Ground` que proporciona los siguientes métodos:

- `getCost()`. Debido a que cada terreno tiene un coste asociado, este método devuelve dicho coste.
- `getGround(int cost)`. Se trata de un método estático que dado un coste, devuelve el terreno al que pertenece. Si el coste que se pasa como argumento no se corresponde con ningún terreno, devuelve `null`.

También podemos observar que cada uno de los equipos tiene un manantial donde los personajes pueden curarse y a donde van cuando estos mueren.

Para diseñar el mapa, se ha hecho uso de la herramienta `Tiled Map Editor` [4] que proporciona una sencilla forma de diseñar mapas por capas, permitiendo que puedas crear capas de distintos tipos (de tiles, de objetos, etc), lo que nos facilita la creación e inicialización de los distintos grids que utilizamos. Lógicamente, las capas que se encuentran en el mapa, se corresponden con los distintos tipos de terrenos mencionados anteriormente (excluyendo la capa de objetos).

Para introducir el mapa diseñado en el proyecto, `LibGDX` nos proporciona la clase `TiledMap`. Sin embargo, esta clase, a la hora de renderizar, si existe alguna capa de objetos por defecto, no la dibuja. Por este motivo, implementamos la clase `TiledMapIADeVProject`, que sobreescribe los métodos de renderizado para que se dibujen las capas de objetos.

Una vez que tenemos el mapa diseñado e introducido en nuestro proyecto, hay que inicializar los distintos grids en correspondencia con dicho mapa. Para ello, está la clase `MapsCreatorIADeVProject`, que proporciona un único método estático `createMaps()` y que se encarga de recorrer, para cada terreno, su capa correspondiente e inicializar los grids a los valores correspondientes.

6 Interacción con el usuario

Todo lo referente a esta sección, se encuentra dentro del paquete `com.mygdx.iadevproject.userInteraction` del proyecto. Para la interacción del sistema con el usuario, se ha hecho uso de la interfaz `InputProcessor` proporcionada por la librería LibGDX. Esta interfaz proporciona método que se llaman cada vez que, por ejemplo, se pulsa una tecla o se clicka con el ratón en la pantalla.

La implementación de la interacción se ha hecho siguiendo una máquina de estados. Debido a que el usuario puede realizar ciertas acciones con los personajes seleccionados, para cada tipo de acción, se ha creado un estado y dentro de ese estado, dependiendo de la tecla pulsada, se hará una acción u otra. El por qué se ha realizado de esta manera es porque hay ciertos comportamientos que requieren que el usuario seleccione a otro personaje; en una situación normal, el programa se quedaría esperando a que el usuario seleccione el personaje. Sin embargo, si se hace eso, el juego se paraliza completamente, incluso el manejo de eventos de la librería, por lo que quedar a la espera de que el usuario seleccione un personaje, no es factible.

Con una máquina de estados, sí podemos controlar que, cuando el usuario haya realizado la acción que se espera, se pase a otro estado que la aplique, sin tener que parar la ejecución del programa.

Los estados que contiene esta máquina son los siguientes:

- **No hay personajes seleccionados**. Refleja el estado de que el usuario no ha seleccionado ningún personaje (estado inicial).
- **Se acaba de seleccionar personajes**. Refleja el estado de que el usuario acaba de seleccionar personajes.
- **Aplicar comportamientos accelerados**.
- **Aplicar comportamientos delegados**.
- **Aplicar un pathfinding**.
- **Aplicar comportamientos en grupo**.
- **Aplicar otros comportamientos**.
- **Realizar una formación**.

Si alguna de las teclas que el usuario pulsa es una de las teclas que manejan el funcionamiento del programa (como puede ser parar el juego, mover la cámara, etc), la máquina no cambia de estado, pues esas teclas se manejan de manera independiente: la máquina de estados es solamente para que el usuario realice las acciones con los personajes.

Para encapsular los comportamientos y acciones que el usuario pueda hacer, se ha creado la clase `UserInteraction` que es la encargada de proporcionar con métodos estáticos dichas acciones. También se encarga de mostrar los mensajes con las posibles acciones que puede realizar el usuario.

Parte IV
IA reactiva

7 Steerings

En el ámbito de los videojuegos, un sistema steering (o sistema de dirección, es español) es un mecanismo que *propone movimientos* a los agentes involucrados en el videojuego en base al entorno local que les rodea, es decir, usando la información del mundo que los agentes captan a través de sus sentidos. Los steerings concretos que se han estudiado y que, por tanto, han sido implementados es este proyecto son: **Steering Uniforme** y **Steering Uniformemente Acelerado**.

Un Steering Uniforme solamente maneja velocidad lineal y velocidad angular, puesto que se da por hecho que en el movimiento no va a intervenir ningún tipo de aceleración (en decir, la aceleración es 0). A partir de dicha información contenida en el steering, el agente modificará su posición y orientación. Por el contrario, un Steering Uniformemente Acelerado sí contiene una aceleración lineal y una aceleración angular, puesto que en este tipo de movimientos sí intervienen aceleraciones (aceleración constante). A partir de esta información contenida en el steering, el agente modificará su velocidad lineal y angular. Al modificar la velocidad lineal y angular, la posición y orientación del agente también será modificada en consecuencia.

7.1 Interfaz Steering

Durante el desarrollo del proyecto, nos ha parecido conveniente el poder manejar todos los steerings de manera uniforme independientemente de la clase de steering concreto. Esto ha sido posible gracias a ciertas características del lenguaje Java como son las clases abstractas, la herencia o la ligadura dinámica. Teniendo esto en cuenta, hemos implementado una clase abstracta vacía (clase `Steering`), tal que todos los steerings concretos heredarán de ella.

7.2 Steering Uniforme

Este tipo de steering ha sido implementado en la clase `Steering_NoAcceleratedUnifMov` y, tal y como se ha comentado, hereda de la clase abstracta `Steering`. En esta clase podemos encontrar dos atributos llamados `velocity` (de tipo `Vector3`) y `rotation` (de tipo float). El primero hace referencia a la velocidad lineal (el vector velocidad) y el segundo hace referencia a la velocidad angular (un escalar). En esta clase también están presentes los métodos *get* y *set* correspondientes a ambos atributos. Además de estos métodos, también hay otro método denominado `getSpeed`, que devuelve el módulo del vector `velocity`.

Es muy importante remarcar que, es este ámbito, *velocity* y *speed* no significan lo mismo. El primero hace referencia al **vector** velocidad, mientras que el segundo hace referencia al **módulo** de dicho vector (es un escalar).

7.3 Steering Uniformemente Acelerado

Este tipo de steering ha sido implementado en la clase `Steering_AcceleratedUnifMov` y, tal y como se ha comentado, hereda de la clase abstracta `Steering`. Esta clase tiene dos atributos llamados `lineal` (de tipo `Vector3`) y `angular` (de tipo `float`). El primero corresponde con la aceleración lineal (un vector) y el segundo con la aceleración angular (un escalar). También se han implementado los métodos *get* y *set* correspondientes a ambos atributos.

8 Comportamientos

Todo lo referente a esta sección, se encuentra dentro del paquete `com.mygdx.iadevproject.aiReactive.behaviour` del proyecto. Para generalizar el uso de los comportamientos, se ha decidido crear la interfaz `Behaviour` que tienen que implementar todos los distintos tipos de comportamientos. Esta interfaz proporciona un solo método:

- `getSteering()`, que no recibe ningún parámetro y devuelve un `Steering`. Este método no recibe ningún parámetro para generalizar los comportamientos, pues no todos los comportamientos necesitan lo mismo para poder calcular su steering. De esta manera, en la creación del comportamiento concreto, se ha de proporcionar todos los datos necesarios para que él pueda calcular su steering, implementando a su manera, este método.

A continuación, en las siguientes subsecciones se muestran todos los comportamientos que hemos implementado.

8.1 Comportamientos no acelerados

Todo lo referente a esta sección, se encuentra dentro del paquete `com.mygdx.iadevproject.aiReactive.behaviour.noAcceleratedUnifMov` del proyecto. Los distintos comportamientos no acelerados que hemos implementado son:

- `Arrive_NoAccelerated`. Este comportamiento consiste en llegar hacia un objetivo, rodeado por un radio de satisfacción en el que se supone que el personaje ya ha llegado, a la máxima velocidad y en un tiempo determinado. Así, para la creación de este comportamiento, son necesarios los siguientes parámetros:
  - El personaje que va a aplicar el comportamiento.
  - El objetivo al que se quiere dirigir.
  - La máxima velocidad con la que se aplica el comportamiento.
  - El radio de satisfacción.
  - El tiempo que se tarda en llegar al objetivo.
- `Flee_NoAccelerated`. Este comportamiento consiste en alejarse de un determinado objetivo, a la máxima velocidad. Para la creación de este comportamiento, son necesarios los siguientes parámetros:
  - El personaje que va a aplicar el comportamiento.
  - El objetivo del que se quiere alejar.
  - La máxima velocidad con la que se aplica el comportamiento.
- `Seek_NoAccelerated`. Este comportamiento es el opuesto al anterior, consiste en ir hacia un determinado objetivo, a la máxima velocidad. Los parámetros son los mismos que los necesarios en el anterior.
- `Wander_NoAccelerated`. Este comportamiento consiste en moverse de manera aleatoria, con un ángulo máximo de rotación a la máxima velocidad. Para la creación de este comportamiento, son necesarios los siguientes parámetros:
  - El personaje que va a aplicar el comportamiento.
  - La máxima velocidad con la que se aplica el comportamiento.
  - El máximo ángulo de rotación que puede cambiar un personaje su orientación.

Es importante destacar, que estos cuatro comportamientos, dentro del método `getSteering()` modifican la orientación del personaje.

8.2 Comportamientos acelerados

Todo lo referente a esta sección, se encuentra dentro del paquete `com.mygdx.iadevproject.aiReactive.behaviour.acceleratedUnifMov` del proyecto. Los distintos comportamientos acelerados que hemos implementado son:

- `Align_Accelerated`. Este comportamiento consiste en adoptar la misma orientación que otro personaje (personaje destino) mediante un movimiento giratorio. La condición principal para llevar a cabo dicho movimiento es que debemos girar hacía el lado cuyo ángulo hacia la orientación destino sea menor. Además, en función del valor del ángulo diferencia (diferencia entre las orientaciones del origen y del destino), la fuente se comportará del manera distinta. Hasta llegar a un ángulo exterior, la fuente gira con la máxima velocidad angular permitida. Al superar dicho ángulo, el personaje va adaptando su velocidad angular. Cuando se supera un ángulo interior, se devuelve un steering con aceleración lineal nulo y con aceleración angular inversa a la velocidad angular del personaje fuente. Para crear este comportamiento son necesarios los siguientes parámetro:
  - Personaje origen y `WorldObject` destino.
  - Máxima aceleración angular que el personaje puede aplicar.
  - Máxima rotación (velocidad angular) que el personaje puede aplicar.

  - `targetRadious` del personaje destino (ángulo interior). En este comportamiento no es un radio, sino un ángulo.
  - `slowRadious` del personaje destino (ángulo exterior). En este comportamiento no es un radio, sino un ángulo.
  - `timeToTarget` es el tiempo que queremos que tarde el personaje en realizar este comportamiento.

- `AntiAlign_Accelerated`. Este comportamiento consiste en adoptar la orientación opuesta de un personaje destino mediante un movimiento giratorio. Al igual que antes, debemos girar hacia el lado cuyo ángulo hacia la orientación deseada sea menor. El funcionamiento del comportamiento es igual que el anterior (aunque en este caso, debemos posicionarnos con orientación inversa a la del destino). Para crear este comportamiento son necesarios los mismos parámetros que el comportamiento anterior.

- `Arrive_Accelerated`. En este comportamiento se definen 3 zonas distintas en las que el comportamiento del personaje es distinto. Podemos encontrar 2 radios ficticios con respecto al personaje destino. En primer lugar, el personaje fuente va aplicando la máxima aceleración establecida hasta llegar al radio exterior. Al llegar al radio exterior, el personaje adapta su velocidad en función de la distancia entre el personaje destino y él, hasta llegar al radio interior. Finalmente, cuando el personaje esta en el radio interior, este comportamiento devuelve el steering nulo (con valor 0). Para crear este comportamiento son necesarios los siguientes parámetro:

  - Personaje origen y `WorldObject` destino.
  - El valor de la máxima aceleración permitida.
  - El valor de la máxima velocidad permitida.
  - `targetRadious`. El radio de la circunferencia ficticia interior.
  - `slowRadious`. El radio de la circunferencia ficticia exterior.
  - `timeToTarget` es el tiempo que queremos que tarde el personaje en realizar este comportamiento.

- `Arrive_Accelerated_WithOneRadius`. Este comportamiento es una mezcla entre un Seek y un Arrive (hereda del comportamiento Seek Acelerado). En este comportamiento podemos encontrar solamente un radio ficticio alrededor del personaje destino. La fuente se mueve aplicando un comportamiento Seek Acelerado normal hasta llegar al radio y cuando entra en él empieza a frenar (devolviendo un steering cuyo vector aceleración tiene un sentido contrario al vector velocidad del personaje fuente). A parte de los parámetros del padre (comportamiento Seek), también son necesarios los siguientes parámetros:

  - `targetRadious`. El radio de la circunferencia ficticia.

- `Flee_Accelerated`. Cuando un personaje aplica este comportamiento, se aleja del personaje destino aplicando la máxima aceleración establecida. Para crear este comportamiento son necesarios los siguientes parámetro:

  - El personaje que va a aplicar el comportamiento.
  - El objetivo al que se quiere dirigir.
  - La máxima aceleración con la que se aplica el comportamiento.

- `Seek_Accelerated`. Este comportamiento consiste en ir hacia un punto objetivo a la máxima aceleración posible. Para la creación de este comportamiento, son necesarios los siguientes parámetros:

  - El personaje que va a aplicar el comportamiento.
  - El objetivo al que se quiere dirigir.
  - La máxima aceleración con la que se aplica el comportamiento.

  Para este comportamiento, se ha hecho uso de las dos implementaciones que nos han proporcionado los profesores en la teoría: la implementación de Millington y la implementación de Reynolds. De esta manera, por defecto, se utiliza la implementación de Millington, pero si se quiere cambiar de implementación, proporciona el método `setMode()` que recibe como parámetro el modo al que se quiere cambiar. El valor del modo de cada una de las implementaciónes está en las constantes `SEEK_ACCELERATED_MILLINGTON` y `SEEK_ACCELERATED_REYNOLDS` correspondientemente.

- `VelocityMatching_Accelerated`. Este comportamiento consiste en, dado un objetivo, ponerse a la misma velocidad que él. Para la creación de este comportamiento, son necesarios los siguientes parámetros:

  - El personaje que va a aplicar el comportamiento.
  - El objetivo al que se quiere ajustar a su velocidad.
  - La máxima aceleración a la que se puede aplicar el comportamiento.
  - El tiempo que en que se alcanza la velocidad del objetivo.

  Este comportamiento, es uno de los comportamientos que puede mostrar las líneas de debug para visualizar su correcto funcionamiento.

8.3 Comportamientos delegados

Todo lo referente a esta sección, se encuentra dentro del paquete `com.mygdx.iadevproject.aiReactive.behaviour.delegated` del proyecto. Los distintos comportamientos delegados que hemos implementado son:

- `CollisionAvoidance`. Este comportamiento consiste en evitar colisiones con objetos en movimiento. Este comportamiento supone que todos los objetivos, así como el personaje, están protegidos por un círculo que los envuelve y detecta colisión cuando el círculo del personaje interseca con alguno de los círculos de los objetivos. Para la creación de este comportamiento, son necesarios los siguientes parámetros:

  - El personaje que va a aplicar el comportamiento.
  - La lista de objetivos a evitar.
  - La máxima acceleración que se aplica para evitar el choque.

  Para evitar las colisiones, el comportamiento tiene en cuenta tanto la velocidad del personaje como la velocidad de todos los objetivos, y con ella calcula si en un futuro van a chocar. De todos los objetivos con los que pueda chocar, se queda con el que esté más cercano; es decir, con el que va a chocar antes. Una vez que lo ha calculado, entonces hace uso del comportamiento `Evade` para evitar chocar con el objetivo obtenido. Es importante destacar que este comportamiento, para evitar las colisiones, no tiene en cuenta los objetos formación pues estos sirven como ancla para la formación y no son un personaje físico como tal (se explican en la sección 10.4).

  Este comportamiento, es uno de los comportamientos que puede mostrar las líneas de debug para visualizar su correcto funcionamiento.

- `Persue`. Este comportamiento hereda de Seek y, al igual que él, se aplica para ir hacia/perseguir a otro personaje (personaje destino). La diferencia con respecto al comportamiento Seek es que en este caso no vamos hacia el propio personaje destino, sino hacia **una predicción** de dónde estará el personaje destino pasado un tiempo. Por tanto, en primer lugar, se realiza dicha predicción y, a continuación, llamamos al comportamiento padre estableciendo como objetivo un personaje ficticio con la posición predicha (en vez del propio personaje real). El único parámetro adicional necesario para crear este comportamiento es `maxPrediction`, que define el tiempo máximo en segundos para predecir la posición del personaje destino.

- `Evade`. Este comportamiento, al igual que el Flee, se aplica para huir/escapar de otro personaje (personaje destino). La diferencia con respecto al comportamiento Flee es que en este caso no escapamos del propio personaje destino, sino de **una predicción** de dónde estará el personaje destino pasado un tiempo. El funcionamiento y parámetros son los mismos que el comportamiento anterior.

- `Face`. Este comportamiento consiste hacer que el personaje se quede mirando hacia un objetivo. Es un tipo de `Align_Accelerated`, por lo que recibe los mismos parámetros que éste. Sin embargo, a diferencia del Align, este comportamiento utiliza el objetivo que se le pasa como parámetro para saber la posición en la que se encuentra y así, poder calcular la orientación a la que se tiene que alinear el personaje.

- `LookingWhereYouGoing`. Este comportamiento consiste en cambiar la orientación del personaje para que mire hacia donde va; es decir, que mire en la dirección a la que se mueve. Es un tipo de `Align_Accelerated` y para su creación, recibe los mismos parámetros que este, exceptuando el objetivo, pues este es calculado por el comportamiento. Su funcionamiento consiste en obtener la orientación del vector velocidad del personaje y alinearse con él.

- `PathFollowingWithoutPathOffset`. Este comportamiento consiste en seguir una serie de puntos aplicando un comportamiento Seek hacia cada uno de ellos (este comportamiento hereda de Seek Acelerado). Este comportamiento tiene 2 modos de funcionamiento: seguir la lista de puntos y para al final (`MODO_PARAR_AL_FINAL`) o seguir la lista de puntos, volver a seguirla haciendo el camino inverso, volver a seguirla... y así hasta el infinito (`MODO_IDA_Y_VUELTA`). Para poder llevar a cabo este segundo modo, los punto de la lista de entrada no pueden eliminarse sin más, sino que debemos de tener otra lista en la que vayamos almacenando aquellos puntos por los que ya hemos pasado (al final, esta segunda lista estará en orden inverso). A la hora de comprobar si se ha alcanzado un punto, no se comparará si la posición actual del personaje es exactamente igual a la posición del punto, sino que existe un margen que viene definido por el parámetro `radius` (es decir, un personaje ha llegado a un punto si está dentro del círculo cuyo radio viene definido por el valor de dicho parámetro). A la hora de aplicar el Seek a cada uno de los puntos de la lista de entrada, se crea un personaje ficticio cuya posición es el punto al que debemos ir. Para la creación de este comportamiento, son necesarios los siguientes parámetros (además de los necesarios para el `Seek_Accelerated`):
  - La lista de puntos por los que el personaje debe pasar.
  - `radius`. El radio de *satisfacción* o *permisividad*.
  - El modo de funcionamiento del comportamiento.
- `PathFollowingWithoutPathOffset_Arrive`. Este comportamiento funciona exactamente igual que el anterior, excepto que en este caso hereda del Arrive en vez de del Seek. La única diferencia relevante a tener en cuenta es que en el comportamiento anterior teníamos un atributo llamado `radius`. En este caso no lo tenemos, puesto que ese radio será el `targetRadious` del padre (del Arrive). En cuanto a los parámetros, no introduce nada nuevo con respecto a los parámetro propios del comportamiento `Arrive` acelerado y a la lista de puntos y el modo (que también estaban presentes en el otro PathFollowing).
- `WallAvoidance`. Este comportamiento consiste en evitar colisiones con otros objetos. La diferencia entre este comportamiento y el anterior, es que la colisión no se detecta cuando el círculo que envuelve al personaje interseca con el círculo de alguno de los objetivos, sino que desde el personaje se lanzan tres rayos (separados un ángulo y longitud determinada), donde el rayo central sigue la dirección de la velocidad del personaje, y si alguno de esos rayos interseca con algún objetivo, entonces lo evita. Es un tipo de `Seek_Accelerated`, por lo que recibe los mismos parámetros que él, exceptuando el objetivo, pues este es calculado por el comportamiento. Para la creación de este comportamiento, son necesarios los siguientes parámetros (además de los necesarios para el `Seek_Accelerated`):
  - La lista de objetivos a evitar.
  - La distancia mínima de separación al objetivo. Debe ser mayor que el radio del círculo que envuelve el personaje.
  - El ángulo de separación entre el rayo central y los laterales.
  - La longitud del rayo central. Por defecto, se establece la longitud de los rayos laterales como el 75% de la longitud del central. Si se quiere modificar esto, tiene otro constructor con el que se puede indicar la longitud de los rayos laterales de manera independiente.

  Para detectar las colisiones entre los rayos y los objetos, se hace uso de la clase `Ray` y el método estático `intersectRaySphere()` proporcionado por la clase `Intersector` (proporcionadas por la librería LibGDX), que además de indicar si hay colisión, devuelve el punto de intersección en el caso de que haya una colisión. En el caso de que haya varios rayos que intersequen con el mismo objetivo, se obtiene aquella intersección que esté más próxima al personaje. Así, una vez que se tiene el punto de intersección, y teniendo en cuenta que suponemos que todos los objetos están envueltos por un círculo, el cálculo de la normal en el punto de intersección es inmediato. Tras tener la normal, calculamos el punto al que realizar el Seek y delegamos en él para mover el personaje hacia esa posición.

  Es importante destacar, que previo a la comprobación de la intersección, se obtiene de la lista de objetivos a evitar, aquel objetivo que esté más cerca del personaje. De tal manera, que no se hace la comprobación de la intersección para todos los objetivos, si no solamente para el objetivo más cercano. También destacar que, al igual que el comportamiento `CollisionAvoidance`, este comportamiento no tiene en cuenta los objetos de tipo formación.

  Este comportamiento, es uno de los comportamientos que puede mostrar las líneas de debug para visualizar su correcto funcionamiento.

- `Wander_Delegated`. Este comportamiento consiste en moverse de manera aleatoria, igual que el `Wander_NoAccelerated` pero de manera acelerada, lo que proporciona un movimiento mucho más suave. Es un tipo de `Face`, por lo que recibe los mismos parámetros que este, exceptuando el objetivo, pues este se calcula dentro del comportamiento. Para la creación de este comportamiento, son necesarios los siguientes parámetros, además de los necesarios para el `Face`:
  - La distancia desde el personaje al Facing.
  - El radio del círculo del Facing.
  - El máximo ángulo que el personaje puede girar.
  - La orientación del personaje de la que se parte.
  - La máxima acceleración a la que se va a mover el personaje.

  Este comportamiento, es uno de los comportamientos que puede mostrar las líneas de debug para visualizar su correcto funcionamiento.

8.4 Comportamientos de grupo

Todo lo referente a esta sección, se encuentra dentro del paquete `com.mygdx.iadevproject.aiReactive.behaviour.group` del proyecto. Los distintos comportamientos de grupo que hemos implementado son:

- `Cohesion`. Este comportamiento consiste en movernos hacia el centro de masas de un conjunto de objetivos con una aceleración máxima (este comportamiento hereda del Seek Acelerado). Solamente se tendrán en cuenta aquellos objetos que estén dentro de un radio (atributo `threshold`). A parte de los parámetros necesarios para el Seek, se necesitan los siguientes parámetro:
  - La lista de objetivos a tener en cuenta para calcular el centro de masas (solo se tendrán en cuenta aquellos que estén dentro del radio establecido).
  - `threshold`. Radio dentro del cual un elemento de la lista de objetivos se tendrá en cuenta.
- `Separation`. Este comportamiento consiste en alejarnos de un conjunto de objetivos aplicando una fuerza de repulsión y con una aceleración máxima establecida. Al igual que antes, solamente se tendrán en cuenta aquellos objetivos que estén dentro de un radio (atributo `threshold`). El vector aceleración lineal final se calculará teniendo en cuenta cada una de las repulsiones calculadas con respecto a cada uno de los elementos que se encuentren dentro del radio establecido. Cabe destacar que este comportamiento **no se ha implementado exactamente como en las transparencias**. Según las transparencias, cada uno de los vectores individuales se deben ir añadiendo al resultado final, sin embargo, esto no nos funcionó (el personaje fuente pasaba de largo sin sufrir ningún tipo de repulsión). Lo que hemos hecho es ir añadiendo **cada vector, pero con sentido contrario** al resultado final. De esta manera sí conseguimos que el behavior haga lo que tiene que hacer. Para la creación de este comportamiento, son necesarios los siguientes parámetros:
  - Personaje origen y la lista de objetivos (tipo `WorldObject`).
  - `threshold`. Radio dentro del cual un elemento de la lista de objetivos se tendrá en cuenta.
  - `decayCoefficient`. Fuerza de repulsión.
  - `maxAcceleration`. Máxima aceleración establecida.

9 Árbitros

Todo lo referente a esta sección, se encuentra dentro del paquete `com.mygdx.iadevproject.aiReactive.arbitrator` del proyecto. Para generalizar el uso de los árbitros, se ha decidido crear la interfaz `Arbitrator` que tienen que implementar todos los distintos tipos de árbitros. Esta interfaz proporciona un solo método:

- `getSteering()`, que recibe como parámetro un objeto del tipo `Map<Float, Behaviour>` que contiene el conjunto de comportamientos (con sus valores de importancia correspondientes) del que se quiere obtener un steering final.

Dependiendo del tipo de árbitro que utilicemos, este método devolverá un steering u otro. Los distintos tipos de árbitros que se han implementado son los siguientes:

- **Árbitro de mezcla ponderada**. Este tipo de árbitro, lo que hace es obtener un steering final como resultado de la mezcla de todos los steerings obtenidos por el conjunto de comportamientos, de manera ponderada. Es decir, para cada comportamiento, se obtiene su steering y el resultado del mismo se añade al steering final multiplicado por el valor de importancia asociado al comportamiento (de ahí que se reciba un objeto de tipo `Map<Float, Behaviour>` que para cada comportamiento se tiene su valor de importancia asociado). Debido a que hay dos tipos de steerings, como se ha comentado en la sección 7, se han implementado dos tipos de árbitros de mezcla ponderada: uno para los comportamientos que devuelve steerings acelerados (clase `WeightedBlendArbitrator_Accelerated`) y otro para los comportamientos que devuelven steerings no acelerados (clase `WeightedBlendArbitrator_NoAccelerated`). El funcionamiento es el mismo en ambos casos, solamente que si un comportamiento devuelve un steering del otro tipo, no se tiene en cuenta.

  Ambos árbitros, en su constructor, reciben como parámetros la máxima aceleración, tanto lineal como angular (en el caso del acelerado), y la máxima velocidad, tanto lineal como angular (en el caso del no acelerado) que dicho árbitro tiene permitido devolver. Por lo que, una vez que se ha calculado el steering final, se comprueba que el valor del steering no supere ninguno de los valores anteriores.

- **Árbitro de prioridad**. Este tipo de árbitro considera que el conjunto de comportamientos recibido como parámetro se encuentra ordenado por prioridad, es decir: el primer objeto del conjunto es el que tiene mayor prioridad. Así pues, recorre dicho conjunto, obteniendo para el comportamiento actual su steering y si este steering es válido (es decir, es distinto de `null` y supera un determinado valor `epsilon`), el árbitro directamente devuelve este steering y termina. Si resulta que de todo el conjunto de comportamientos, ninguno es válido porque no ha superado el valor `epsilon`, el árbitro devuelve, en este caso, el steering obtenido por el último comportamiento del conjunto, independientemente del valor que tenga.

  A diferencia del árbitro anterior, con este árbitro se puede usar comportamientos que devuelvan tanto steerings acelerados como no acelerados; lo que implica que el árbitro puede devolver cualquier tipo de steering. El determinado valor `epsilon` es un valor que se le pasa al árbitro en su constructor, y que indica el valor mínimo que un steering debe de tener para que se considere válido.

10 Modelo

Esta es una de las partes más importantes del videojuego, puesto que en ella podemos encontrar a los **agentes** que intervendrán e interactuarán en el transcurso del juego. Concretamente, en este paquete podemos encontrar la implementación de los objetos del mundo (concepto general que lo engloba todo), la implementación de los personajes, la implementación de los obstáculos y la implementación de las formaciones (que han sido tratadas como un tipo especial de personaje).

10.1 Clase WorldObject

La clase `WorldObject` es un concepto general que representa a una entidad del juego. Todas las entidades concretas (descritas en los siguientes subapartados) heredarán de esta **clase abstracta**. Una cosa muy importante a tener en cuenta es que esta clase abstracta hereda a su vez de la clase `Sprite` de la librería libgdx. Esto se ha hecho así para aprovechar todas las características y propiedades que ya posee la clase `Sprite`, como por ejemplo, la posición, la orientación, la anchura, la altura, las funciones de dibujo u otra funcionalidad ya preprogramada en la librería.

Solamente hay que tener en cuenta una cosa muy importante: a lo que nosotros llamamos orientación, la librería lo llama rotación. Esto a su vez puede llevar a confusiones con la velocidad angular (que para nosotros es la rotación). Para evitar todos estos problemas de nomenclatura y todas las confusiones que esto pueda provocar, se han tomado ciertas medidas:

- Se han creado el par de funciones `getOrientation` y `setOrientation` que llaman a las funciones correspondiente del padre (`super.getRotation` y `super.setRotation`).

- Se han sobrescrito las funciones `getRotation` y `setRotation` heredadas del padre para que si se llaman en el hijo (en la clase `WorldObject`), lancen una excepción. Estas funciones no se deben llamar nunca, puesto

que si deseamos consultar o modificar la orientación de una entidad, deberemos hacer uso de las funciones `getOrientation` y `setOrientation` de la clase `WorldObject`.

Estas modificaciones solucionan el problema y evitan que se puedan producir confusiones con los nombres y errores de concepto a lo largo del resto del proyecto.

En cuanto a los atributos propios de la clase `WorldObject` (los no heredados de la clase `Sprite`), podemos encontrar la velocidad del `WorldObject` (vector velocidad de tipo `Vector3`), la velocidad angular del `WorldObject` (escalar de tipo float) y la velocidad máxima del `WorldObject` (escalar de tipo float que hace referencia a la máxima velocidad que puede tener independientemente de su comportamiento). Este último atributo se ve reflejado en el método `setVelocity`. Si el módulo del vector que se pasa como parámetro supera a la velocidad máxima permitida, entonces el vector velocidad que se asigna al `WorldObject` es un vector con la misma dirección y sentido que el pasado como parámetro, pero con un módulo igual al atributo `maxSpeed` (máxima velocidad permitida del `WorldObject`).

Cabe destacar que el método de dibujo de la textura del `WorldObject` ha cambiado. Por defecto, la librería libgdx dibuja la textura de tal manera que la esquina inferior izquierda de ésta es la posición del sprite. Sin embargo, lo que nosotros queremos es que la posición del sprite corresponda con **el centro** de la textura. Por ese motivo, el método `draw` de la clase `Sprite` ha sido sobrescrito en la clase `WorldObject`. Antes de proceder al dibujo de la textura, la posición del `WorldObject` se modifica temporalmente, de tal manera que el centro de la textura corresponda con la posición real del `WorldObject`. Tras dibujar la textura, la posición vuelve a restablecerse. Para conseguir esto, basta con restar a la coordenada 'x' e 'y' del `WorldObject` la mitad de la anchura y la mitad de la altura respectivamente.

10.2 Clase Character

La clase `Character` representa a un personaje del videojuego y hereda de la clase `WorldObject`. En esta clase podemos encontrar ciertos atributos propios como la lista de comportamientos del personaje, la formación a la que pertenece el personaje (si no pertenece a ninguna, este atributo es `null`) y el árbitro que gestiona la lista de comportamientos del personaje y selecciona el steering final a aplicar (se describe con más detalle es la sección correspondiente). Cabe destacar que este último atributo es obligatorio. Todos los personajes deben tener un árbitro que selecciona el steering final a aplicar.

La lista de comportamientos es realmente un `Map` en el que se almacenan parejas formadas por el comportamiento propiamente dicho y un valor de tipo float que representa la importancia de ese comportamiento dentro de la lista. Este valor de importancia puede significar distintas cosas en función del árbitro que se esté usando (esto se detallará más en la sección correspondiente). En cualquier caso y si no se indica lo contrario, el comportamiento se añade con una valor por defecto que viene dado por la constante `DEFAULT_IMPORTANCE` presente en esta clase.

Además de lo anterior, en la clase `Character` también podemos encontrar otros métodos relevantes como el método `getNewOrientation`, que dado un steering devuelve la orientación final del personaje tras la aplicación de dicho steering (solo para steerings no acelerados); el método `applyBehaviour`, que llama al árbitro para obtener un steering y aplica dicho steering mediante el método `applySteering`; el método `applySteering`, que aplica el steering que se le pasa como parámetro (llama al método `update` solo si el personaje actual no forma parte de una formación); y el método `update`, que es el que realmente aplica el steering pasado como parámetro, modificando las propiedades del personaje. Este último método comprueba si el steering pasado como parámetro es de tipo uniforme o uniformemente acelerado y actúa en consecuencia.

Una cosa importante a destacar es que **si una entidad pertenece a una formación, no se podrá mover libremente**, es decir, da exactamente igual la lista de comportamientos que tenga, ya que ninguno influirá sobre él. Una entidad perteneciente a una formación se moverá acorde con la formación. Esto se explica más en detalle en el apartado correspondiente.

10.3 Clase Obstacle

La clase `Obstacle` representa a un obstáculo fijo y, al igual que la anterior, hereda de la clase `WorldObject`. La única peculiaridad que tiene esta clase es que todos los métodos como `getVelocity`, `getSpeed` o `setRotation_angularSpeed`, entre otros, están sobrescritos para que se devuelvan 0 o el vector 0. Esto es así porque, como acabo de comentar, esta clase representa obstáculos estáticos que no se mueven.

10.4 Formaciones

La **clase abstracta** `Formation` representa a un conjunto de personajes organizados en formación. Hemos planteado el tema de las formaciones de tal manera que heredan de la clase `Character`, es decir, las formaciones son un tipo especial de personaje. Además, en la clase `Formation` se hace uso del **patrón composite**, lo que va a permitir que uno de los integrantes de una formación pueda ser a su vez otra formación. Esto quiere decir que **se pueden construir formaciones de formaciones todo lo grandes y profundas que deseemos** (con todos los niveles de profundidad que deseemos). De esta clase heredarán todos los tipos de formaciones concretas que se deseen crear.

Además de los atributos heredados del padre, esta clase dispone de algunos atributos propios como la lista de personajes que pertenecen a la formación y la orientación que adoptarán los componentes de la formación. En cuanto al segundo atributo, hay diversas opciones: orientación libre, que todos los componentes miren hacia el interior de la formación, que todos los componentes miren hacia el exterior de la formación o que todos los componentes adopten la misma orientación que la formación (la misma orientación que el objeto `Formation`).

A parte del patrón composite, en esta clase también se ha aplicado el **patrón método plantilla**. Podemos observar este patrón en los métodos `getCharactersPosition` y `getComponentFormationSteerginToApply` (la implementación de ambos métodos se delega a los hijos de la clase `Formation`). El primer método sirve para calcular y obtener las posiciones **con respecto al centro o ancla de la formación** que ocuparán los integrantes de la misma. El segundo método sirve para obtener el steering que será aplicado sobre un componente de la formación.

La implementación del patrón composite se ve reflejada en el método `applySteering` (este método ha sido sobrescrito). Mientras que en el padre (clase `Character`), simplemente se llamaba al método `update`, en esta ocasión se realizan las siguientes acciones (solo si la formación no pertenece a otra formación):

- En primer lugar, se aplica el método `update` sobre la formación, lo que provocará que el objeto formación (el ancla) se mueva según el steering concreto pasado como parámetro.
- A continuación, se calculan las posiciones que ocuparán cada uno de los integrantes de la formación. Para ello, ejecutamos el método `getCharactersPosition`. Cabe destacar que la lista de posiciones obtenida **tendrá la misma longitud que la lista de integrantes de la formación**.
- Seguidamente, se recorre cada componente de la formación y se aplican las siguientes acciones:
    * Se indica momentáneamente que el componente actual no pertenece a ninguna formación. Esto debe hacerse para poder aplicarle un steering y, por tanto, poder aplicarle un movimiento.
    * A continuación, se construye un `WorldObject` ficticio cuya posición es la posición destino del componente actual de la formación (una de las que han sido calculadas anteriormente).
    * Seguidamente, se llama al método `getComponentFormationSteerginToApply` para obtener el steering a aplicar sobre el componente actual de la formación y **se ejecuta el método** `applySteering` **con el steering que se acaba de calcular**. En este paso es donde se puede observar claramente la aplicación del patrón composite. Gracias a esta llamada, conseguimos "mover en profundidad" todo lo que haya en la formación.
    * Finalmente, volvemos a indicar que el componente actual pertenece a una formación, lo que evitará que se pueda mover o manipular desde otro sitio (mientras siga estando en una formación).

Al igual que pasaba en el padre, este método se sigue llamando desde `applyBehaviour`.

A parte de todo lo anterior, en esta clase podemos encontrar un método llamado `drawFormationPoints`. Este método recibe un *renderer* como parámetro y dibuja en él las posiciones que deben ocupar los componentes de la formación.

A parte de la clase abstracta `Formation`, se han implementado diversos tipos de formaciones concretas como formación en circulo, formación en estrella y formación en linea. Tal y como se ha comentado anteriormente, debido a la utilización del método plantilla, cada uno de los tipos concretos de formación deben implementar los métodos `getCharactersPosition` y `getComponentFormationSteerginToApply`.

Hay diversas cuestiones a tener en cuenta en algunos tipos de formaciones concretas implementadas:

- La formación en estrella es un tipo de formación en círculo (hereda de la clase correspondiente a la formación en circulo), en la que ciertos componentes poseen un mayor desplazamiento con respecto al sitio que deberían ocupar en una formación normal en círculo. Además, para que una formación en estrella pueda crearse se deben cumplir ciertas condiciones: que la cantidad de integrantes sea mayor estricto que 3 y que la cantidad de integrantes sea par. Si estas condiciones no se cumplen, los componentes formarán un círculo normal.
- En la formación en linea, el ángulo que forma la linea de componentes está directamente relacionado con la orientación de la propia formación (del objeto formación). Por ese motivo, como en algunos casos la orientación de un `WorldObject` pueda cambiar y oscilar muy rápidamente y bruscamente, no resulta adecuado que el ángulo de la linea de componentes sea igual a la orientación del `WorldObject`, puesto que las posiciones finales de los componentes cambiarían bruscamente y rápidamente y la organización de la formación en línea nunca se llevaría a cabo. La solución a este problema ha sido partir el rango de orientaciones del `WorldObject` en subrangos y asignar a cada subrango un valor del ángulo de la linea de componentes de la formación. Con esto conseguimos que las pequeñas modificaciones de la orientación del objeto `LineFormation` no afecten a las posiciones finales de los componentes de una formación en línea.

A la hora de calcular el steering concreto a aplicar sobre cada uno de los componentes de la formación (en la función `getComponentFormationSteerginToApply`), también se hace uso de un árbitro y de una lista de comportamientos. **Es muy importante no confundir estos elementos con el árbitro y la lista de comportamientos del padre o con el árbitro y la lista de comportamientos del propio objeto Character**. En la función `getComponentFormationSteerginToApply` se declaran y se usan ambos elementos "a pelo", puesto que se llegó a la conclusión de preconfigurar y fijar los comportamientos de los componentes de las formaciones y sólo permitir la libre configuración de la lista de comportamientos de la propia formación. Esto se debe a que realmente, lo que se mueve es la formación y los componentes de ésta deben limitarse a ir a los puntos que han sido calculados para ellos. Para conseguir esto, los componentes de una formación no usarán sus propios comportamientos (los de la clase Character) sino que estarán obligados a hacer todo lo que se les diga (mediante el árbitro y la lista de comportamientos "a pelo").

Otra cosa importante a tener en cuenta es que un objeto de tipo `Formation` no tiene puntos de vida. Aunque realmente sí que tiene dicho atributo (porque lo hereda del padre), éste está inutilizado. Los puntos de salud de una formación serán **la suma de la salud de todos los elementos que la componen**. Del mismo modo, los métodos para añadir y reducir la salud de una formación estarán vacíos (sobreescritos y vacíos).

Parte V
PathFinding

11 PathFinding

Tras haber implementado todos los comportamientos y tipos de movimientos que se pueden aplicar a los personajes de nuestro videojuego, es también muy importante contar con un mecanismo que nos permita obtener el camino adecuado para llegar desde un origen a un destino en función de distintos criterios y teniendo en cuenta cierta información del entorno. Dicho mecanismo se denomina *PathFinding* y es un elemento muy importante en aquellos videojuegos en los que sea necesario (aunque, como también se ha visto en clase, otros videojuegos no lo necesitan, por lo que no lo implementan).

El mecanismo de PathFinding visto en clase se apoya fundamentalmente en 2 estructuras. Por un lado, es totalmente imprescindible la matriz de costes de terrero. Esta estructura es un grid ficticio que se sitúa sobre el mapa físico y en el que se almacenan los costes del terreno del mapa del juego. Por otro lado, también se necesita una matriz de distancias o heurística. En esta matriz se almacenan los costes desde cada una de las celdas hasta una celda objetivo. Ambas matrices deben tener las mismas dimensiones.

Es muy importantes destacar la existencia de 2 tipos de posiciones con las que vamos a trabajar. Por un lado, tenemos la posición real de una entidad en el mapa y, por otro lado, dicha entidad también tendrá una posición en el grid (esta posición indica **los índices** de la matriz que representa al grid). Esta segunda posición es con la que trabaja el PathFinding. Teniendo esto en cuenta, habrá que contar con las funciones necesarias para pasar de un tipo de posición a otro.

A la hora de pasar una posición del mapa a una posición del grid, entre otras cosas, se eliminan los decimales para poder obtener una posición entera (para usar como índice en el grid ficticio). Por tanto, se podría decir que el mecanismo de búsqueda está basado en un **grid cuadrado por vértices**. Sin embargo, cuando volvemos a convertir de una posición del grid a una posición del mapa, nos interesa que el movimiento se realice por el centro del tile para evitar que el personaje se mueva justo por la separación entre un tipo de terreno y otro (justo entre un tile y otro). Por tanto, una de las cosas que se hace al transformar de una posición del grid a una posición del mapa es **sumar la mitad del lado de un tile a ambas coordenadas de la posición final**. De esta forma conseguimos que los puntos finales del mapa siempre estén en el centro de los tiles que representan el terreno.

Para este proyecto, hemos implementado **2 tipos de Pathfinding**. Ambos se basan en el algoritmo LRTA* con espacio de búsqueda minimal. La diferencia fundamental entre ambos es que uno calcula todos los puntos de golpe desde el origen al destino (pathfinding continuo) y el otro va devolviendo punto por punto y va calculando un nuevo punto cada vez que sea necesario (pathfinding punto a punto). El primero de ellos devuelve una lista completa con todos los puntos desde el origen hasta el destino y basta con ejecutarlo tan solo una vez. El segundo de ellos devuelve un único punto (aunque también en una lista) y debe ser ejecutado cada vez que queramos obtener el siguiente punto al que ir. Esto se explicará más detalladamente en el apartado correspondiente.

11.1 Distancias

Tal y como se ha comentado en clase, las heurísticas que se han implementado son: **distancia de Manhattan**, **distancia de Chebyshev** y **distancia Euclídea**. La información concreta y la forma de cálculo de cada una de estas distancias se especifica detalladamente en las transparencias de la asignatura.

En le código fuente de nuestro proyecto podemos encontrar la interfaz `Distance`. Esta interfaz sirve para generalizar todos los tipos de distancia concretos y tratarlos de manera homogénea. El método fundamental de todas las distancias implementadas es `getMatrixOfDistances`. Este método recibe las dimensiones que deberá tener la matriz de salida y la posición objetivo y devuelve la matriz de distancias correspondiente.

A la hora de realizar un PathFinding desde un origen hasta un destino, la posición objetivo que se le pasa al método `getMatrixOfDistances` es la posición destino a la que queremos llegar. Esto quiere decir que la matriz de

distancias calculada almacenará la distancia desde todas las posiciones del grid a la posición destino a la que queremos llegar (posición destino **del grid**).

11.2 Algoritmo LRTA*

Tal y como se dice en la especificación de estas prácticas, se debe implementar el algoritmo LRTA* con espacio de búsqueda minimal. Eso quiere decir que el espacio de búsqueda solamente estará compuesto por el estado actual en el que nos encontramos.

Para el caso del pathfinding continuo, el algoritmo LRTA* recibe como entrada la matriz de costes del terreno, la matriz de distancias (que será distinta según la heurística elegida), la anchura y altura de ambas matrices, la posición origen **del grid** y la posición destino **del grid**. Para el caso del pathfinding punto a punto no se recibirá la posición inicial, sino que cada vez que lo llamemos le pasaremos la posición actual desde la que queremos obtener el siguiente punto (que, efectivamente, al principio será el punto de origen). El algoritmo LRTA* trabajará (modificará) sobre la matriz de distancias y devolverá una lista de puntos que corresponden con todas aquellas posiciones **del grid** por las que hay que pasar para llegar desde el origen hasta el destino (para el caso de pathfinding continuo) o, directamente, el siguiente punto al que debemos ir (para el caso de pathfinding punto a punto). En este segundo caso, obviamente, el algoritmo no realiza ningún bucle.

Para calcular la función *f(x)* de un estado concreto, serán necesarios los siguientes elementos:

- El coste que supone realizar una acción (movernos de una celda a otra). Este valor viene definido por la constante `default_action_cost` de la clase `LRTA_star`.
- La distancia que hay desde el estado actual hasta la posición de destino. Este valor podemos obtenerlo de la matriz de distancias.
- El coste del terreno en la posición o estado actual en el que nos encontramos.

Estos componentes serán la base para obtener el valor de la función *f(x)* de un estado concreto y, por tanto, para que el algoritmo LRTA* calcule el camino adecuado desde un origen hasta un destino.

Además, también se han implementado diversos métodos necesarios para el correcto funcionamiento del algoritmo LRTA*. El primero de ellos es `generateSuccessors`. Este método recibe una posición del grid y devuelve todos sus vecinos (tanto en horizontal, como en vertical, como en diagonal). El segundo método necesario es `getSuccessorWithTheSmallestHeuristic`. Este método selecciona el vecino tal que *f(x)* devuelva el menor valor.

11.3 Clase PathFinding

Esta es la clase principal del mecanismo de PathFinding. Al igual que ocurre con el propio algoritmo LRTA*, cada tipo de pathfinding implementado tiene su propia clase y lo implementa de distinta forma. Las particularidades de cada uno se comentarán en las siguientes secciones.

11.3.1 Pathfinding continuo

En este pathfinding el método `applyPathFinding` es el encargado de realizar la conversión de posiciones del mapa a posiciones del grid (y viceversa) y de la ejecución del algoritmo LRTA*. En este caso, la función `applyPathFinding` devolverá el resultado final (lista con todos los puntos) tras la primera ejecución y, si queremos volver a hacer un pathfinding, deberemos crear y comenzar de nuevo. Este método recibe como parámetro la matriz de costes del terreno, la heurística deseada, el tamaño del lado de celda del grid, la anchura y altura de las matrices, las coordenadas 'x' e 'y' **del mapa** del origen y las coordenadas 'x' e 'y' **del mapa** del destino.

En primer lugar, las coordenadas del mapa son transformadas en coordenadas del grid; seguidamente, se calcula la matriz de distancias; a continuación, se ejecuta el algoritmo LRTA*; y finalmente, se vuelven a transformar todas las coordenadas del grid al mapa (de todos los puntos que ha devuelto el algoritmo LRTA*).

Cabe destacar que al transformar las coordenadas del mapa de los puntos origen y destino a coordenadas del grid se pierde información, puesto que las coordenadas del mapa son de tipo `float` (con decimales) y las coordenadas del grid no tienen decimales (puesto que estas coordenadas son realmente los indices con los que se accederán a ambas

matrices). Por este motivo, al volver a transformar las coordenadas del grid al mapa, se realiza también una segunda transformación. El primer y último punto de la lista que devuelve el algoritmo LRTA* (el origen y destino), son reemplazados por las coordenadas originales que se pasaron como parámetro a la función `applyPathFinding`.

11.3.2 Pathfinding punto a punto

Al contrario que pasaba con el pathfinding anterior, en este caso el constructor de la clase juega un papel más importante. En este caso, es en el constructor donde se almacenan todos los atributos necesarios (según los parámetros pasados al constructor) y donde se realiza la transformación de la coordenada inicial del pathfinding de coordenadas del mapa a coordenadas del grid. Además, puesto que en esta ocasión el pathfinding va a ser llamado más de una vez (cosa que no pasaba en el pathfinding anterior), en el constructor también se crea el objeto que representa el algoritmo LRTA* y se almacena como un atributo de la clase (para ir usando siempre el mismo objeto y no perder el avance realizado).

Un elemento fundamental en este tipo de pathfinding es el atributo `objetivoActual`. Inicialmente, este atributo contiene el punto de partida del pathfinding y, conforme vayamos avanzando, iremos actualizando este atributo con la posición a la que debemos ir. En este caso, el pathfinding punto a punto irá devolviendo siempre la misma posición hasta que no lleguemos a ella (ver las comprobaciones iniciales del método `applyPathFinding`). Una vez que el personaje haya alcanzado el siguiente objetivo actual, el pathfinging y el algoritmo LRTA* continuarán calculando.

11.4 Pathfinding táctico

Para añadir información táctica al pathfinding es suficiente con tener en cuenta el valor de la matriz de influencia (o cualquier otra información que queramos añadir) en el cálculo de la función f(x) en el algoritmo LRTA*. La utilización de información táctica se puede activar o desactivar sobre la marcha. Para ello, hemos añadido los flags correspondientes en las clases del pathfinding y del algoritmo LRTA* y las funciones necesarias para activar y desactivar dichos flags.

Concretamente, nuestro pathfinding táctico podrá usar la información táctica proveniente del mapa de influencia y la información táctica propia de cada rol. Dicha información táctica en relación a cada rol concreto se puede encontrar en el método `getTacticalCost` de las clases `Archer` y `Soldier`. Cada rol concreto podrá tener un coste diferente según cada uno de los terrenos presentes en el mapa.

Parte VI
IA táctica

12 Modificaciones en el modelo

Para incluir la parte táctica hemos tenido que añadir cierta funcionalidad y atributos en la clase `Character`, así como crear un enumerado que refleje los equipos posibles en el videojuego.

12.1 Clase Character

Las modificaciones que se han hecho dentro de la clase `Character` explicada en la sección 10.2 han sido las siguientes:

- Se ha añadido el atributo estático `DEFAULT_HEALTH` que refleja la vida por defecto que tienen todos los personajes. Este atributo está inicializado a 20.000.
- Se ha añadido un atributo de tipo `TacticalRole` (ver sección 15) que contiene el rol del personaje.
- Se ha añadido un atributo de tipo `Team` que contiene el equipo al que pertenece el personaje.
- Se han añadido dos atributos para controlar la vida del personaje: uno para la vida actual y otra para la vida máxima que puede tener un personaje.
- Debido a que puede haber interacción con el usuario y que si un personaje es seleccionado, este debe hacer caso a lo que indica el usuario, se ha añadido un atributo booleano que indica si el rol de un personaje está habilitado o no.
- Debido a que ahora es el rol del personaje el que le indica qué comportamientos tiene que realizar, la aplicación de los comportamientos sigue siendo la misma, pero en vez de llamar directamente al método `applyBehaviour()`, previamente hay que actualizar el rol para que actualice la lista de comportamientos del personaje (ver sección 15 donde se explican los roles). Para esto, se ha creado el método `updateTacticalRole()` que sencillamente lo que hace es, si el rol está activo (no ha sido seleccionado por el usuario) actualizarlo (siempre y cuando el personaje no pertenezca a una formación) y después llamar al método `applyBehaviour()` tal y como se explica en la sección 10.2.
- Un personaje, por defecto, no tiene por qué tener un rol. Para asignarle un rol, se ha creado el método `initializeTacticalRole()` que recibe como parámetro un objeto del tipo `TacticalRole` y que lo que hace es asignar el rol al personaje e inicializarlo.
- Para que, dependiendo del rol del personaje, este se mueva más o menos rápido dependiendo del terreno que esté pisando, se ha creado el método `getVelocityFactorOfThisCharacter()` que devuelve un valor entre 0 y 1, que indica cómo se aplica la velocidad obtenida del steering obtenido al personaje (método `update()`). Si el personaje no tiene rol o está en una formación que no tiene rol, este método siempre devolverá un 1, es decir, no se verá afectada la velocidad del personaje con respecto al terreno que esté pisando. Si el personaje no tiene rol, pero pertenece a una formación con rol, entonces, este método devolverá lo que devuelva el método `getVelocityFactor()` del rol de la formación. Sin embargo, si tiene rol, este método devolverá lo que devuelva el método `getVelocityFactor()` del rol (ver sección 15).
- También se han creado dos métodos para reducir y aumentar la vida del personaje, que se usarán en los comportamientos de ataque y curación. Estos métodos reciben como parámetro la vida que se quiere reducir/aumentar. La vida del personaje se visualiza siempre y cuando este no pertenezca a una formación (en este caso, se visualizará solamente la vida de la formación), para ello, también se ha creado otro método que dibuja la vida del personaje al lado del mismo.
- Como se ha comentado, que el usuario seleccione a un personaje, implica que su rol se deshabilite y el personaje haga solamente lo que el usuario le mande. De igual manera, cuando el usuario lo deseleccione, el personaje debe volver a aplicar su rol como si no hubiera pasado nada. Para ello, se han creado dos métodos, uno para indicar que el personaje ha sido seleccionado y otro para indicar que el personaje ha sido deseleccionado.

- Por último, para que se vayan mostrando el estado táctico por el que pasa el personaje, también se ha implementado un método que recibe como parámetro una cadena con el estado en el que está (además de los objetos necesarios para dibujar). Este método se llamará en todos los estados/nodos de los roles tácticos para ir indicando qué es lo que está realizando el personaje.

12.2 Enumerado Team

Para poder reflejar los equipos del videojuego, se ha creado el enumerado `Team` que contiene tres equipos: `FJAVIER, LDANIEL y NEUTRAL`. Como se puede apreciar, los nombres de los equipos los hemos puesto en honor a los dos profesores de la asignatura. El equipo de `FJAVIER` es el que se sitúa en la base de abajo a la derecha, mientras que el equipo de `LDANIEL` es el que sitúa en la base de arriba a la izquierda. El equipo `NEUTRAL` es un equipo neutral y se añadió para poder poner otros personajes en el videojuego que no pertenezcan a ningún equipo (tal como animales o cosas por el estilo). Sin embargo, en el diseño final no hemos incluido personajes neutrales, pero así, ya está todo preparado por si se quieren introducir.

Este enumerado tiene un único método interesante: `getEnemyTeam()` que devuelve el equipo contrario al objeto que realiza el método: si el equipo es `LDANIEL` devolverá `FJAVIER` (y viceversa), pero si es `NEUTRAL` devolverá `NEUTRAL`.

13 Otros comportamientos

A parte de todos los comportamientos descritos anteriormente, para la parte táctica también se han implementado algunos otros comportamientos. Realmente, no son comportamientos como tal, sino que son diversas acciones o procesos para los cuales ha resultado conveniente el poder tratarlos homogéneamente y como si fueran comportamientos normales. Estas acciones son el **ataque** y la **cura**. Cabe destacar que estos comportamientos **devolverán el steering nulo**, puesto que, realmente, lo importante de estos comportamientos no es el comportamiento propiamente dicho, sino todas las funciones/comprobaciones que se realizan dentro de él (en el método `getSteering`).

13.1 Comportamiento de ataque

Este comportamiento representa la orden o la acción de atacar a un objetivo. Por tanto, los elementos que este comportamiento necesita como entrada son: el personaje que realiza el ataque, el personaje que recibe el ataque, el daño que realiza el personaje origen al personaje destino (salud que se le resta al personaje destino) y la máxima distancia a la que se puede realizar el ataque.

En primer lugar, se comprueba si la distancia a la que se encuentran origen y destino es menor o igual que la distancia máxima permitida para realizar el ataque. En caso afirmativo, se llamará al método `reduceHealth` de la clase `Character`, para reducir la vida al target.

Cuando un personaje (no formación) realiza un ataque, este comportamiento será añadido a su lista de comportamiento. Tal y como se puede observar, el hecho de tratar el mecanismo de ataque como un comportamiento más, nos permite poder añadirlo a la lista de comportamientos de un personaje y poder tratarlo de manera homogénea con el resto de comportamientos.

13.2 Comportamiento de cura

Este comportamiento representa el proceso de cura (incremento de salud) por parte de un personaje. Por tanto, los elementos que este comportamiento necesita como entrada son: el personaje fuente cuya salud va a ser incrementada y el valor de salud a incrementar.

En este caso, lo único que se hace en el método `getSteering` es llamar al método `addHealth` de la clase `Character`. Con esto conseguimos incrementar el nivel de salud de un personaje fuente.

Al igual que antes, cuando un personaje (no formación) se cura, este comportamiento será añadido a su lista de comportamientos.

13.3 Ataque y cura en las formaciones

A la hora de entender los procesos de ataque y cura para el caso de las formaciones, hay que tener muy claro lo siguiente:

- Cuando un conjunto de personajes están en una formación, la lista de comportamientos propia de cada personaje no se tiene en cuenta. Por el contrario, se usará la lista de comportamientos del propio objeto formación (para mover el ancla de la formación) y un conjunto de comportamientos establecidos directamente "a pelo" en el método correspondiente para hacer que los componentes de la formación vayan al punto que les corresponde.
- Los objetos formación como tal **ni atacan ni se curan**. Los que atacan y se curan son los elementos de la formación.
- Del mismo modo, los objetos de tipo formación no pueden ser atacados (a la hora de atacar se comprueba que el objetivo no sea una formación) ni se pueden curar (ya que, de nuevo, los que se curan son los componentes de la formación).
- Teniendo en cuenta los 2 puntos anteriores, los comportamientos de ataque y cura jamás podrán estar en la lista de comportamientos de un objeto de tipo formación, ya que estas acciones pueden ser realizadas exclusivamente por personajes como tal.
- Los componentes de la formación no tienen estructura táctica (máquina de estados o árbol de decisión). Esta estructura solamente se encuentra en el objeto tipo formación (en la "raíz", en caso de haber formaciones de formaciones).

Sabiendo todo esto, es necesario contar con algún mecanismo para que el objeto tipo formación (el ancla) pueda comunicar a los integrantes de la formación que deben curarse o atacar a un objetivo determinado. Esta comunicación se realiza a través de los métodos `enableAttackMode` y `disableAttackMode` (para el caso del comportamiento de ataque) y los métodos `enableCure` y `disableCure` (estos métodos se encuentran en la clase `Formation`). Al llamar a estos métodos lo que hacemos es modificar los flags correspondientes y almacenar otros atributos necesarios para los comportamientos de ataque y cura. Teniendo en cuenta esos flags, en el método `getComponentFormationSteerginToApply` de cada tipo de formación concreta se añadirá el comportamiento de ataque o cura a la lista de comportamientos establecida "a pelo" que, como sabemos, se usará para controlar a cada uno de los componentes de la formación. De esta manera conseguimos el poder imponer el comportamiento de ataque o cura a cada uno de los elementos que componen una formación.

14 Acciones y comprobaciones

Debido a que la implementación de la parte táctica de los personajes debe ser lo más expresiva posible, decidimos crear una serie de métodos que nos sirvieran de puente con la parte reactiva. De esta manera, por ejemplo, haciendo uso del método `notCollide()` un personaje ya tiene los comportamientos necesarios para que evitar las colisiones, independientemente de si está usando un comportamiento u otro. Ejemplos de algunos métodos implementados son: atacar a un enemigo, atacar al enemigo más cercano, curarme, ir hacia un determinado punto (aplicando un pathfinding), etc. Todos estos métodos (y más), nosotros los hemos llamado Acciones y se encuentran dentro del paquete `com.mygdx.iadevproject.checksAndActions` en la clase `Actions`.

De igual manera que para obtener los comportamientos correspondientes a una determinada acción, también decidimos crear una serie de métodos que nos sirvieran de puente para la obtención de información por parte de un personaje. Por ejemplo, el método `amINearFromMyBase()` comprueba si el personaje se encuentra cerca de su base. Ejemplos de algunos de estos métodos son: comprobar si estoy en la base enemiga, si un personaje es del equipo contrario, si un personaje ha muerto, si ha recuperado toda su vida, etc. Todos estos métodos (y más), nosotros los hemos llamado Comprobaciones y se encuentran dentro del mismo paquete anterior en la clase `Checks`.

En esta documentación no se exponen todas las comprobaciones y todos las acciones implementadas (la gran mayoría se muestran en los diagramas de los roles específicos). Si se quiere saber más sobre ellos, se encuentran documentados en el JavaDoc generado del proyecto.

15 Roles Tácticos

La implementación de los roles tácticos la hemos realizado a través de la interfaz `TacticalRole` que contiene dos métodos importantes:

- Método `initialize()`. Este método realiza la inicialización de la estructura táctica que se va a utilizar. Por ejemplo, en el caso de que un rol se implemente con una máquina de estados, pues crear e inicializar dicha máquina.
- Método `update()`. Este método realiza la actualización de la estructura táctica inicializada y las acciones pertinentes dependiendo de dicha actualización.

Ambos métodos reciben como parámetro el personaje al que se va a aplicar los comportamientos obtenidos tras la inicialización/actualización de la estructura táctica.

También contiene los siguientes métodos:

- `getVelocityFactor()`. Devuelve el factor de velocidad que un rol tiene para un determinado terreno. Este método devolverá un valor entre 0 y 1 que se usa para modificar la velocidad que se aplica a un personaje antes de aplicar un determinado steering. Con este método, se permite que los personajes, dependiendo de su rol, vayan a una velocidad dependiendo del terreno por el que vayan.
- `getTacticalCost()`. Devuelve el coste táctico que un rol tiene asociado a un determinado terreno.
- `getMaxDistanceOfAttack()`. Devuelve la máxima distancia de ataque de un rol.
- `getDamageToDone()`. Devuelve el daño que puede hacer al atacar un rol.
- `getMaxSpeed()`. Devuelve la máxima velocidad a la que puede ir un rol.

Los roles tácticos que hemos implementado nosotros han sido: soldado y arquero, tanto ofensivos como defensivos. Tanto los ofensivos como los defensivos, tienen los mismos valores para los métodos anteriores (dependiendo de si son arqueros o soldados). La diferencia entre ellos está en la estructura con la que se han implementado: los roles ofensivos (tanto soldados como arqueros) se han implementado con un árbol de decisión; mientras que los roles defensivos (tanto soldados como arqueros) se han implementado con una máquina de estados. En las siguientes subsecciones, se comentará más en profundidad sobre cada uno de estos roles.

En esta interfaz también podemos encontrar una constante denominada `health_cure`. Esta constante indica el ritmo con el que todas las unidades (independientemente del rol concreto) recuperan su vida cuando se están curando.

Todo esto se encuentra dentro del paquete `com.mygdx.iadevproject.aiTactical.roles` del proyecto.

15.1 Roles defensivos

Ambos roles defensivos (arquero y soldado) se han implementado como una máquina de estados. Para ello, se ha hecho uso de la interfaz `StateMachine` proporcionada por la librería LibGDX [2]. Esta máquina de estados hace uso de la interfaz `State` proporcionada también por la librería LibGDX, que proporciona los siguientes métodos:

- `enter()` que se llama cada vez que se entra al estado.
- `update()` que se llama cada vez que la máquina de estados se actualiza y este es el estado actual de la máquina.
- `exit()` que se llama cuando se sale del estado.
- `onMessage()` que se llama si la entidad recibe un mensaje del despachador de mensajes mientras está en este estado.

Estos métodos reciben como parámetro una entidad con la que se puede trabajar en cada método. En nuestro caso, esta entidad será un objeto de la clase `Character`.

La interfaz `StateMachine` proporciona varios métodos para poder realizar distintas acciones con la máquina de estados. De entre ellos, las que hemos utilizado son:

- `setInitialState()`. Se emplea para establecer el estado inicial de la máquina, que se le pasa como parámetro.
- `isInState()`. Comprueba si la máquina está en el estado pasado como parámetro.
- `changeState()`. Cambia el estado actual al estado pasado como parámetro.
- `update()`. Actualiza la máquina: esto implica llamar al método `update()` del estado actual.

Al crear la máquina de estados, esta pide que se indique el objeto "propietario" de la máquina de estados; esto es, el objeto que la máquina pasa como parámetro en los métodos de la interfaz `State`. Para introducirle los estados que nosotros queremos a la máquina de estados, hemos tenido que crear clases concretas que implementaran la interfaz `State` anteriormente mencionada. Para todos los estados, la entidad que recibe como parámetro es de la clase `Character`.

Por último, en las Figuras 2 y 3 se muestran las máquinas de estados concretas para los soldados y arqueros defensivos. La idea que hay detrás de estos roles es que ambos tienen que patrullar una zona (los soldados patrullan su base, los arqueros patrullan los waypoints de los puentes), hasta que se encuentran a un enemigo cerca y lo atacan, siempre y cuando no estén lejos de su base/waypoint. Como se trata de un rol defensivo, estos van a estar atacando hasta que se mueran. Por último, debido a que hay 6 waypoints para cada patrullar, si hay más arqueros defensivos que waypoints, estos se quedarán a la espera (haciendo cosas aleatorias) de que uno se libere para cogerlo. Tanto en el caso del arquero como en el soldado, cuando están yendo hacia un lugar determinado, o en el caso del arquero cuando está a la espera de reservar un waypoint, si se acerca cualquier enemigo o si le atacan, el personaje no hará caso y seguirá haciendo su acción correspondiente. El motivo principal de esta decisión es que al tratarse de roles defensivos, su misión es defender tanto la base como los waypoints de los puentes. Por lo que su prioridad es ir hacia esos puntos y una vez allí atacar a todos los enemigos que se acerquen.

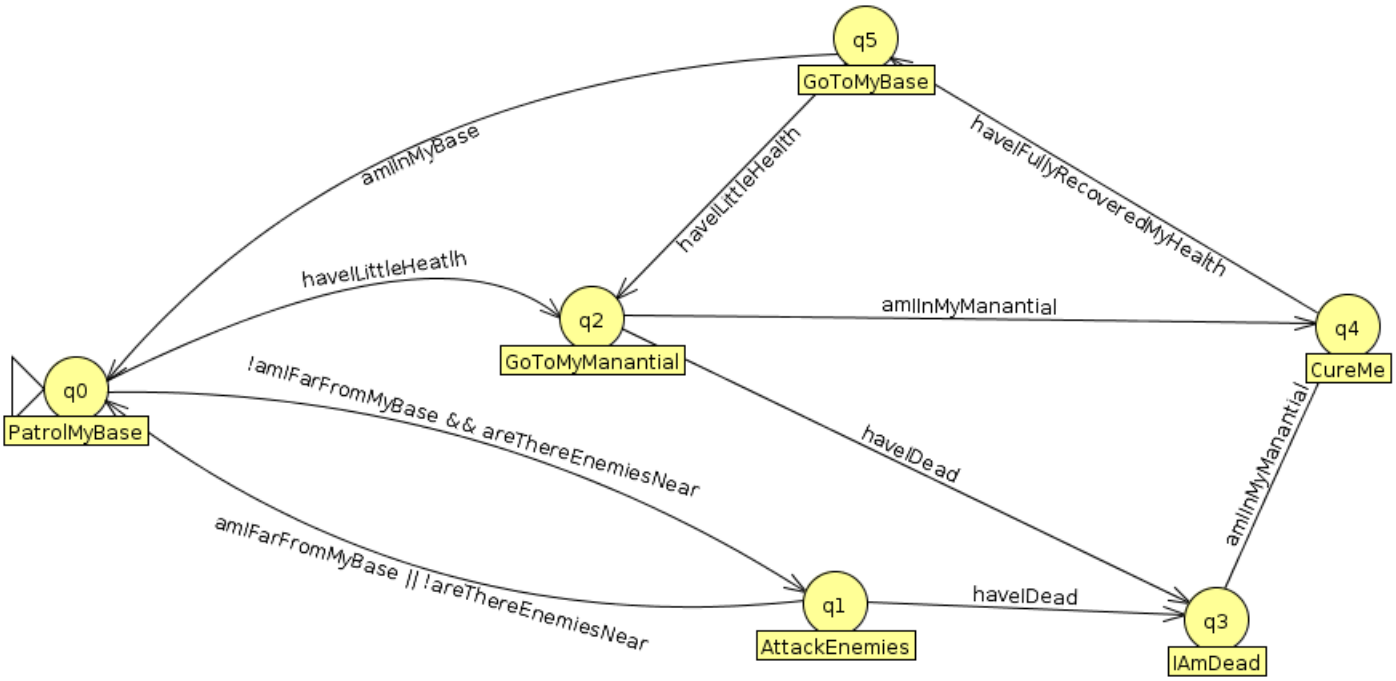

Figura 2: Máquina de estados para el soldado defensivo.

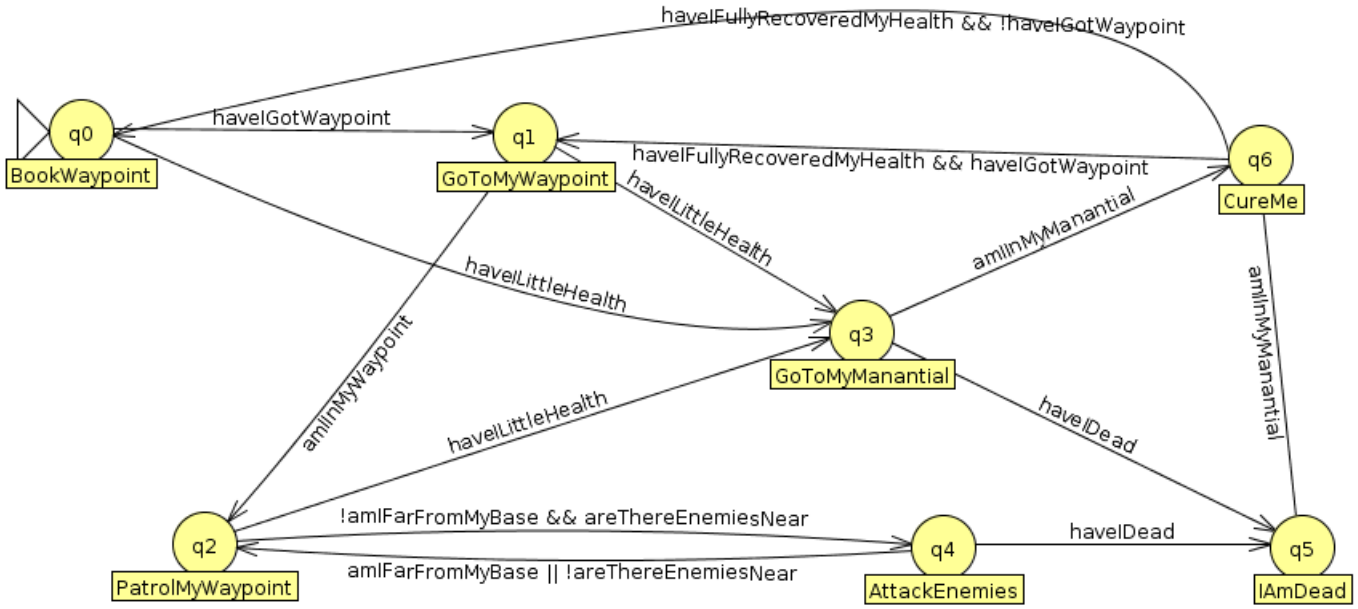

Figura 3: Máquina de estados para el arquero defensivo.

Es importante destacar los siguientes puntos:

- Si no se cumplen las condiciones necesarias en el estado actual, no se cambia de estado. De ahí que no haya transiciones en las figuras sobre los propios estados. Cuando no se cambia de estado, exceptuando el estado `AttackEnemies` y los que requieren el uso del Pathfinding, la máquina de estados no se actualiza, ya que cuando se entró al estado, se calcularon los comportamientos correspondientes y el personaje sigue haciendo lo mismo, por lo que no es necesario calcularlo de nuevo.

- Los estados que requieran el uso del Pathfinding (`GoToMyBase, GoToMyManantial y GoToMyWaypoint`) para evitar que se esté calculando el Pathfinding cada vez que se actualice el estado, hacemos uso del método `enter()` del estado para que cada vez que se entre se calcule el Pathfinding, mientras que en el método `update()` lo que se hace es aplicar el Pathfinding para que nos dé el siguiente punto a seguir.

- En el estado `AttackEnemies` se ataca siempre al enemigo más cercano (de ahí que se tenga que actualizar el estado si no hemos cambiado). También el personaje intenta alejarse del objetivo a una distancia en la que él pueda atacar. Así pues permitimos que aquellos personajes que tengan una distancia de ataque mayor (los arqueros en nuestro caso), puedan sacar provecho de ello cuando se ataca. También se mira al personaje que se ataca.

- Debido a que los mecanismos de ataque y cura se han implementado de manera diferente al resto de presonajes normales, los estados `AttackEnemies` y `CureMe` hacen uso del método `exit()` de la interfaz `State` para que, cuando se sale de alguno de estos estados, se realicen las acciones correspondientes a dejar de atacar y curar.

- En el estado `IAmDead` se traslada directamente el personaje a la posición de su manantial, estableciendo su posición a la posición del manantial.

- En el estado `BookWaypoint` mientras que espera a tener un waypoint libre, el personaje lo que hace es moverse de manera aleatoria (aplicando un Wander), para evitar que se quede parado.

- Debido a que los arqueros defensivos tienen que reservar el waypoint que tienen que patrullar, solamente se va a liberar un waypoint cuando el arquero que lo está patrullando muere. Es decir, entrar al estado `IAmDead` implica que el arquero deja libre su waypoint para que otro pueda cogerlo. Sin embargo, cuando un arquero entra al estado `GoToMyManantial` porque quiere curarse, no se libera el waypoint porque no ha muerto.

- Los estados de patrulla `PatrolMyBase y PatrolMyWaypoint`, en ambos casos, consiste en realizar un seguimiento de los waypoints establecidos. En el caso de las bases son waypoints alrededor de la base, mientras que en el caso de los puentes, un arquero patrulla el waypoint que tiene reservado y su vecino.

- Consideramos que de primeras, los soldados se encuentran en la base (o cerca de ella), por lo que en su estado inicial no se hace ningún Pathfinding. En cambio, como los arqueros, mientras que no tienen un waypoint que patrullar, se mueven de manera aleatoria, cuando reserva un waypoint, sí se calcula un Pathfinding para ir a él, porque pueden estar en la otra punta del mapa.

15.2 Roles ofensivos

Tanto el arquero ofensivo como el soldado ofensivo han sido implementados con **árboles de decisión**. Los correspondientes arboles se puede encontrar en las propias clases `OffensiveSoldier` y `OffensiveArcher`. En estas clases podemos observar una serie de atributos que representan cada uno de los nodos hoja a los que el árbol de decisión podrá llegar (en cada nodo hoja se realizarán una serie de acciones). Además de los nodos hojas, también podemos encontrar otro atributo llamado `lastNode`. Aunque estemos tratando con un árbol de decisión y aunque cada vez que lo llamemos se realicen todos las comprobaciones (desde el principio hasta que una se cumpla), es bastante necesario contar con un mecanismo para *recordar* cual es el último nodo hoja al que llegamos. Esto es necesario debido a que al llegar a un nodo hoja a través de las ramas del árbol, en algunos casos será necesario realizar ciertas acciones sobre el nodo anterior en el que nos encontrábamos (protocolo de salida del nodo anterior). Esto solamente se hará en caso de que el último nodo hoja y el actual al que hemos llegado sean distintos.

Además de los atributos mencionados, en las clases `OffensiveSoldier` y `OffensiveArcher` hay un método llamado `update`. Tal y como se ha comentado antes, este método realiza la actualización de la estructura táctica. Para el caso concreto de los árboles de decisión, el método `update` empezará a "'recorrer el árbol" (en profundidad y de izquierda a derecha) hasta que las condiciones pertinentes (checks) se cumplan y podamos llegar a un nodo hoja (esto se ha implementado con sentencias *if-else*).

Una cosa importante a tener en cuenta es que al llegar a un nodo hoja del árbol, antes de ejecutar dicho nodo hoja deberemos comprobar si el nodo al que hemos llegado es distinto de `lastNode` (nodo anterior al que se llegó). En caso afirmativo, se ejecutará el **método de salida del nodo anterior**, a continuación, el **método de entrada** del nuevo nodo hoja al que hemos llegado y, finalmente, ya podremos proceder a la ejecución de dicho nodo hoja (el nuevo nodo hoja se asigna a `lastNode`). Otra cosa importante a tener en cuenta es que los árboles de decisión no requieren inicialización (el método `initialize` está vacío).

Puesto que la finalidad de todo personaje ofensivo es ir a atacar al enemigo a su base (a la base del enemigo) y, también, atacar a todo enemigo que se le ponga por delante, el árbol de decisión de los soldados ofensivos será el mismo que el de los arqueros ofensivos. Las nodos hojas a los que podemos llegar son los siguientes:

- `IAmDead` → Cuando la vida de un personaje llega a 0, éste es trasladado directamente a la posición de su manantial, estableciendo su posición a la posición del manantial. Este nodo no requiere método de entrada ni método de salida. Puesto que en un árbol siempre se ejecutan todas las comprobaciones hasta llegar a un nodo hoja y debido a que este nodo está el primero, cuando movamos el personaje a la fuente, **debemos poner a uno su vida**. Si no hacemos esto, el árbol de decisión se quedará pillado en esta comprobación (porque al tener vida 0, siempre se considerará que estoy muerto) y el personaje no funcionará correctamente (jamás podremos pasar de la primera rama del árbol).

- `GoToMyManantial` → En este nodo le decimos al personaje que tiene ir a curarse a su manantial. Puesto que esto requiere un pathfinding (que será un atributo del nodo), este nodo sí tiene un método de entrada. Este método de entrada crea el objeto pathfinding y lo almacena en el propio nodo.

- `CureMe` → En este nodo debemos decirle al personaje (ya en el manantial) que debe regenerar su nivel de vida. Además, mientras que ese nivel de vida no llegue al máximo, el personaje deberá permanecer en el manantial (aplicando un comportamiento que constantemente lo lleve a la posición del manantial). Para el caso de las formaciones, los mecanismos de ataque y cura se han implementado de una manera diferente al resto de personajes normales (se comentará en el apartado correspondiente), por ese motivo, este nodo va a necesitar un método de salida para indicar este hecho (que los componentes de la formación deben parar de curarse).

- `GoToEnemyBase` → En este nodo se indica al personaje que debe ir a la base enemiga. Al igual que pasaba la acción de ir al manantial, este nodo requerirá un pathfinding y, por tanto, un método de entrada.

- `AttackEnemies` → En este nodo se indica al personaje (de manera apropiada, según sea un personaje normal o una formación) que debe atacar a un determinado objetivo (el enemigo más cercano). Para poder llevar a cabo ese ataque, el personaje se posicionará en la posición adecuada según la distancia desde la que pueda atacar (depende de si es un arquero o un soldado). Al igual que pasa con la cura de los personajes, las formaciones se tratan de una forma distinta y, por tanto, este nodo va a necesitar un método de salida (para indicar a los componentes de la formación que deben parar de atacar).
- `Win` → En este nodo, el equipo al que pertenece el personaje ha ganado. Las condiciones de victoria y como se han gestionado estos elementos se explicará en el apartado correspondiente.

Para generalizar y trabajar de manera homogénea con todos los nodos hoja, se ha creado la interfaz `Node`. Todos los nodos la implementarán.

Hay que tener muy claro que, debido a que en un árbol de decisión siempre se hacen todas las comprobaciones posibles desde el principio hasta llegar a un nodo hoja, hay que tener mucho cuidado en el diseño del árbol y en qué ramas se ponen antes que otra. Si no hacemos esto, podremos llegar a situaciones raras que podrán provocar efectos no deseados sobre el comportamiento táctico del personaje. En la Figura 4 se puede observar una representación del árbol de decisión (que, como se ha comentado antes, es el mismo para los arqueros y soldados ofensivos).

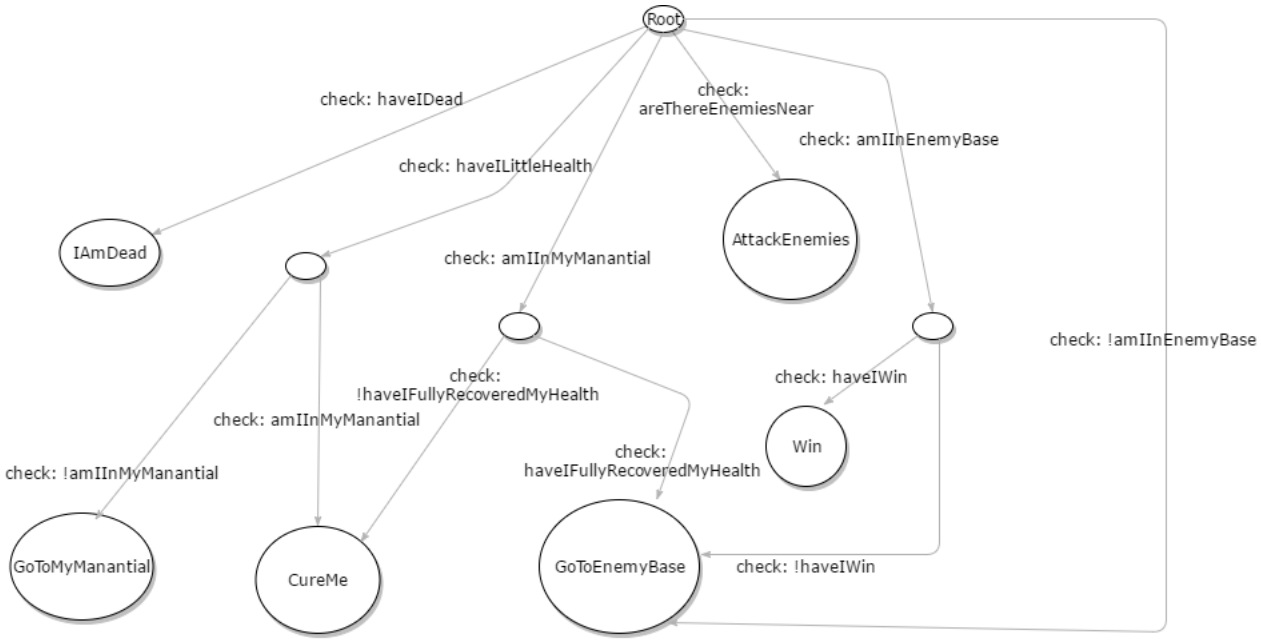

Figura 4: Árbol de decisión para los arqueros y soldados ofensivos.

Tal y como se puede observar, todo aquello que necesite un pathfinding estará lo más a la derecha posible en el árbol.

16 Waypoints

Un Waypoint es una posición preestablecida en el mapa que puede tener cierta finalidad y puede ser usada para un determinado propósito. Estos puntos en el mapa normalmente son establecidos por los propios diseñadores del mapa y pueden tener diversos usos como uso táctico (para este caso, ocupar estas posiciones proporcionaría algún tipo de ventaja táctica) o pueden representar posiciones defensivas (posiciones más adecuadas para situar personajes defensivos). En cualquier caso, estos puntos deben tener asociada cierta información que dependerá del uso que les queramos dar.

Para nuestro caso concreto, vamos a implementar y a usar waypoints con finalidad defensiva, es decir, los waypoints establecidos representarán posiciones interesantes a defender. Toda esta funcionalidad se encuentra en el fichero

`Waypoints` del paquete `com.mygdx.iadevproject.waypoints`.

Cada equipo tendrá 2 conjuntos de waypoints:

- Waypoints de la base → Estos waypoints serán usados para defender la base. Los personajes cuyo rol sea soldado defensivo harán un Pathfollowing a la lista de puntos correspondientes a su base.
- Waypoints de los puentes → Cada equipo dispone de 6 waypoints de este tipo. Tal y como se ha explicado en el apartado correspondiente, en el mapa hay 4 puentes, aunque solo 3 de ellos serán defendidos por los personajes (cuyo rol sea arquero defensivo) de cada equipo. Para cada puente de esos 3, un equipo tendrá 2 waypoints en su lado correspondiente del puente (lo que harán un total de 6 waypoints para cada equipo). Al igual que antes, un personaje aplicará un Pathfollowing para patrullar su lado correspondiente de los puentes (Pathfollowing con una lista de 2 puntos).

Para repartir de manera ordenada los waypoints de los puentes y evitar que pueda haber aglomeración de personajes en un mismo puente, se ha implementado un sistema de reserva y liberación de waypoints (solamente para los waypoints de los puentes). Como solamente hay 6 waypoints en los puentes en cada equipo, solamente podrá haber 6 arqueros defensivos patrullando los puentes (2 del mismo equipo en cada puente). Si otros personajes intentan reservar un waypoint de los puentes, no podrán y pasarán a moverse de manera aleatoria (mediante el comportamiento Wander).

En la clase correspondiente y para el caso de los waypoints de los puentes, podemos encontrar las siguientes estructuras:

- `bridgesWayPoints_team_X` (una para cada equipo) → Esta estructura es de tipo `Map<Vector3, ValueOfBridgeWaypoint>` se inicializa al principio de la aplicación y contiene la siguiente información:
  - La clave (de tipo Vector3) del Map es el waypoint del puente que será reservado por el personaje.
  - El valor (de tipo ValueOfBridgeWaypoint) del Map está formado a su vez por 2 elementos: un booleano que indica si ese waypoint está ocupado o no (inicialmente todos estarán a false) y un Vector3 donde se almacena el waypoint vecino (el otro waypoint del mismo lado del mismo puente).

  Esta estructura se inicializará al principio del juego y, durante la partida, solamente se modificará el atributo booleano, pero no se añadirán ni se eliminarán elementos del Map.
- `bridges_CharacterAndWaypointAssociation_team_X` (una para cada equipo) → Esta estructura es de tipo `Map<Character, Vector3>`, inicialmente está vacía y contiene la siguiente información:
  - La clave (de tipo Character) del Map es el personaje que ha reservado un determinado waypoint.
  - El valor (de tipo Vector3) del Map corresponde con el Waypoint reservado por el personaje.

  En esta estructura se almacenará qué personaje ha reservado un determinado waypoint y se irá modificando a lo largo de la partida.

Hemos decidido trabajar con este tipo de estructuras para que el proceso de reserva y liberación de los waypoints de los puentes sea lo más rápido y eficiente posible. Cabe destacar, que los personajes del equipo neutral no pueden tener acceso a estas estructuras ni a esta funcionalidad.

Cuando un personaje va a reservar un waypoint, comprobamos que el personaje no tenga ya uno asociado (indexando al personaje en el Map `bridges_CharacterAndWaypointAssociation_team_X` de su equipo). Si sí tiene un waypoint ya asociado, simplemente devolvemos ese y su vecino. Si no lo tiene, debemos recorrer la lista `bridgesWayPoints_team_X` de su equipo en busca de un waypoint libre. Si no hay waypoints libres, devolvemos la lista vacía. En caso contrario, se reserva el waypoint para dicho personaje. Esa reserva implica que el booleano de ocupación del waypoint pasará a valer true y a la lista de asociaciones de personaje-waypoint se le añadirá una nueva entrada. Tras dicho proceso de reserva, devolveremos una lista con el waypoint reservado y con su vecino en el puente (en el mismo lado del puente).

Cuando un personaje libera un waypoint, lo primero que comprobamos es si, efectivamente, dicho personaje tenía asociado un waypoint. En caso afirmativo, modificamos el booleano de ocupación del puente (vuelve a valor false) y eliminamos la asociación entre el personaje de entrada y el waypoint que tenía reservado.

Además de todos los métodos comentados, en esta clase también podemos encontrar diversos métodos para dibujar los waypoints de las bases de los puentes. Estos métodos podrán servir para mostrar información adicional e información de depuración y, además, para poder comprobar si el personaje está siguiendo el camino que debería seguir.

17 Puntos de moral

Para implementar las condiciones de victoria del videojuego, hemos usado *puntos de moral*. Cada equipo (cada base, realmente) tendrá una puntuación que inicialmente tiene valor 1 e irá aumentando o disminuyendo según distintos sucesos. Cuando hay personajes de un equipo en su base y no hay personajes del equipo contrario, los puntos de moral de esa base aumentarán hasta llegar al nivel máximo. Cuando hay personajes de un equipo en la base contraria y no hay ningún personaje del otro equipo en esa misma base (es decir, la base no está protegida por personajes de ese equipo), los puntos de moral de esa base decrecerán hasta llegar a 0. Ganará el equipo que consiga reducir la moral de la base contraria a 0. También puede darse el caso de que simultáneamente los puntos de moral de ambas bases lleguen a 0. En ese caso, se producirá un empate.

En la clase `MoralPoints` del paquete `com.mygdx.iadevproject.checksAndActions` podemos encontrar los siguientes elementos:

- Atributo `moralPointsByDefault` → Indica la cantidad máxima de puntos de moral que pueden tener los equipos.
- Atributo `moralPointsSubtractedByCharacter` → Indica la cantidad de puntos de moral que cada personaje sustrae de la base enemiga.
- Atributo `moralPointsAddedByCharacter` → Indica la cantidad de puntos de moral que cada personaje añade a su propia base. Es lógico que este valor sea mayor que el de arriba.
- Atributo `moralPoints_base_X` → Cantidad actual de puntos de moral de un equipo determinado. Al principio de la partida, ambos equipos tiene 1 punto de moral (que se irá incrementando hasta llegar a nivel máximo).
- Métodos para añadir, reducir y resetear los puntos de moral del equipo de un determinado personaje (se pasa como parámetro).
- Método para mostrar/dibujar los puntos de moral en cada una de las bases.

En el método `haveIWin` de la clase `Checks` se comprueba, precisamente, si la base del equipo contrario del personaje que se le pasa como parámetro ha llegado a 0 puntos de moral. En caso afirmativo, el equipo correspondiente al personaje pasado como parámetro ganará la partida. Cuando en el árbol de decisión de los personajes ofensivos llegamos al nodo `Win` (porque el check haveIWin ha devuelto true), lo que hacemos es modificar un flag que se encuentra en la clase principal `IADeVProject` (una variable global). Ese flag se usará para comprobar si un equipo ha ganado y parar el juego.

Tal y como hemos dicho antes, la moral de una base subirá si hay personajes de ese equipo y no hay personajes del equipo contrario. Para conseguir esto, el método `addMoralPointsToMyBase` es llamado en el estado `patrolMyBase` (en la clase `DefensiveSoldier`). En ese estado solamente nos encontramos cuando patrullamos nuestra base (porque no hay enemigos cerca). Cuando un enemigo se acerque a mi base, cambiaremos de estado (para atacar a ese enemigo) y, por tanto, ya no se seguirán incrementado los puntos de moral de la base.

Del mismo modo, tal y como hemos dicho antes, la moral de un base bajará si hay personajes de un equipo en la base contraria y no hay ningún personaje del otro equipo en esa misma base (la base está desprotegida). Para conseguir esto, el método `subtractMoralPointsToEnemyBase` es llamado en el método `amIInEnemyBase` de la clase `Checks`. Cuando hacemos esta comprobación ya estamos seguros de que no hay enemigos cerca, ya que la comprobación de si hay enemigos cerca se encuentra antes en el árbol de decisión (más a la izquierda). Del mismo modo, cuando sí haya enemigos cerca (aunque también estemos en la base contraria), los atacaremos y no llegaremos a la comprobación `amIInEnemyBase`, por lo que ya no se seguirán sustrayendo puntos de moral a la base enemiga.

18 Mapas de influencia

Los mapas de influencia reflejan la presencia o *poder* de cada uno de los equipos en una zona o casilla concreta del mapa. Para cada personaje del equipo, la influencia de ese equipo comenzará desde la casilla actual en la que se encuentra el personaje y podrá extenderse una determinada distancia (una cantidad determinada de casillas alrededor del personaje). Aunque, es habitual que conforme mayor sea la distancia menor sea la influencia que ese personaje va teniendo sobre el terreno.

Tal y como hemos planteado los mapas de influencia, cada equipo tendrá su propio mapa de influencia y el contenido de cada uno de esos mapas se calculará en función de la posición de los personajes de ese equipo (solamente los de ese equipo). Esta separación inicial en diversos mapas nos servirá para disponer y controlar de una forma más exhaustiva la presencia y valores concretos de cada uno de los personajes de cada equipo. Cuando vayamos a dibujar la influencia final sobre el terreno, sí que se tendrán en cuenta todos los mapas de todos los equipos. Es importante tener en cuenta que, aunque cada equipo tenga su propio mapa, **el equipo con mayor influencia en una casilla, controlará dicha casilla** (esto se tendrá en cuenta al dibujar el mapa).

A la hora de calcular la influencia que ejerce un personaje concreto sobre el terreno que le rodea, son necesarios 2 valores: el máximo valor de influencia (que será la influencia que se ejercerá sobre la casilla en la que se encuentra el personaje) y la máxima distancia de influencia (a partir de esa distancia, el personaje no ejercerá influencia sobre el terreno). Conforme nos vamos alejando de la casilla actual en la que se encuentra el personaje, **el valor de influencia va disminuyendo en 1**. El mecanismo de cálculo es la distancia de Chebyshev (tal y como se ha visto en la parte de pathfinding, aunque con algunas modificaciones). Teniendo en cuenta cual es el resultado de esta distancia, hemos considerado que aplicarla en este caso es la mejor opción y lo más conveniente. A la hora de modificar la matriz de influencia de un equipo, solo se tendrá en cuenta la región afectada por un personaje concreto (de acuerdo a la distancia máxima establecida), es decir, que para cada personaje no se recorrerá la matriz entera. Cabe destacar que un personaje neutral no ejercerá influencia, no tendrá ninguna matriz asociada y no podrá tener acceso a esta funcionalidad.

Cuando hay varios personajes del mismo equipo cerca, a la hora de calcular el mapa de influencia de ese equipo, puede ocurrir que el valor de una celda supere el valor máximo permitido (ya que las influencias individuales se van sumando). Cuando esto ocurre simplemente sustituimos dicho valor por el máximo permitido.

Esta funcionalidad consta básicamente de los siguientes métodos:

- En primer lugar, deberemos inicializar todas las variable necesarias usando los métodos habilitados para ello. Esto solo se hace 1 vez.
- A continuación, en cada iteración del bucle del juego debemos llamar al método `updateSimpleMapOfInfluence` para actualizar la información de los mapas de influencia. Cabe destacar que solamente se tendrán en cuenta los personajes como tal y no los objetos de tipo formación.
- Finalmente, debemos llamar al método de dibujo para poder visualizar la influencia que ejerce cada equipo sobre el terreno. O bien en forma de mini mapa o bien sobre el propio mapa completo (en función de los parámetros que pasemos a la función).

Para realizar el dibujo del mapa de influencia calculado, existe el método `drawInfluenceMap`. Este método recibe los siguientes parámetros:

- renderer.
- output_grid_cell_size → Tamaño **de salida** del lado de cada cuadradito o celda del mapa de influencia.
- positionX y positionY → Posición de la esquina inferior izquierda del mapa de influencia.
- filled → Booleano que indica si las celdas del mapa de influencia se rellenarán o no. Si este flag es falso, solamente se colorearán los lados de las celdas.

Ajustando el tamaño de celda y las posiciones del mapa podemos conseguir que el mapa de influencia se muestra como un minimapa junto al mapa real o sobre el propio mapa real (con los cuadrados no rellenos, en este caso).

A la hora de dibujar el mapa, cada equipo tiene distintos tonos de azul y distintos tonos de rojo (un equipo es el color azul y otro es el rojo). Además, el color blanco representará una casilla neutral. Para cada posición del mapa de influencia, a parte de comprobar si la celda debe rellenarse/colorearse o no, también se harán otras comprobaciones:

- Si el mapa de costes de terreno en esa posición corresponde con un terreno infranqueable, no se dibujará la influencia.
- Si la influencia de un equipo es mayor a la del otro, se dibujará del color correspondiente al equipo con mayor influencia y del tono de color correspondiente. Para obtener el todo de color correspondiente **se resta el valor del equipo con mayor influencia menos el valor del equipo con menor influencia** (puesto que aunque un equipo controle una casilla, la influencia del otro equipo también afecta). Ese resultado se divide entre 5 (porque hay 5 tonos de color por cada equipo) y se usa el tono correspondiente al cociente obtenido.
- Finalmente, si el parámetro `filled` está a true (y si no hemos entrado por ninguna de las comprobaciones anteriores), se dibujará la celda de color blanco (celda sin influencia). Esto se usa para que, en el caso del minimapa, sí aparezca el fondo blanco, pero en el caso de dibujar el mapa de influencia sobre el mapa real, solamente aparezcan las influencias de los equipos y no se dibujen las zonas sin influencia.

A la hora de integran esta información con el pathfinding táctico de un personaje, habría que pasar el mapa de influencia **del equipo contrario**. El pathfinding debe tener en cuenta la influencia del equipo rival (coste añadido) para NO IR por esas zonas. Por tanto, a priori no tiene sentido usar el mapa de influencia de mi propio equipo, ya que estaría penalizando las zonas controladas por mí mismo.

Para finalizar, voy a adjuntar una captura de pantalla de nuestro videojuego en la que se puede observar el mapa de influencia final dibujado (tanto en forma de minimapa como sobre el mapa real):

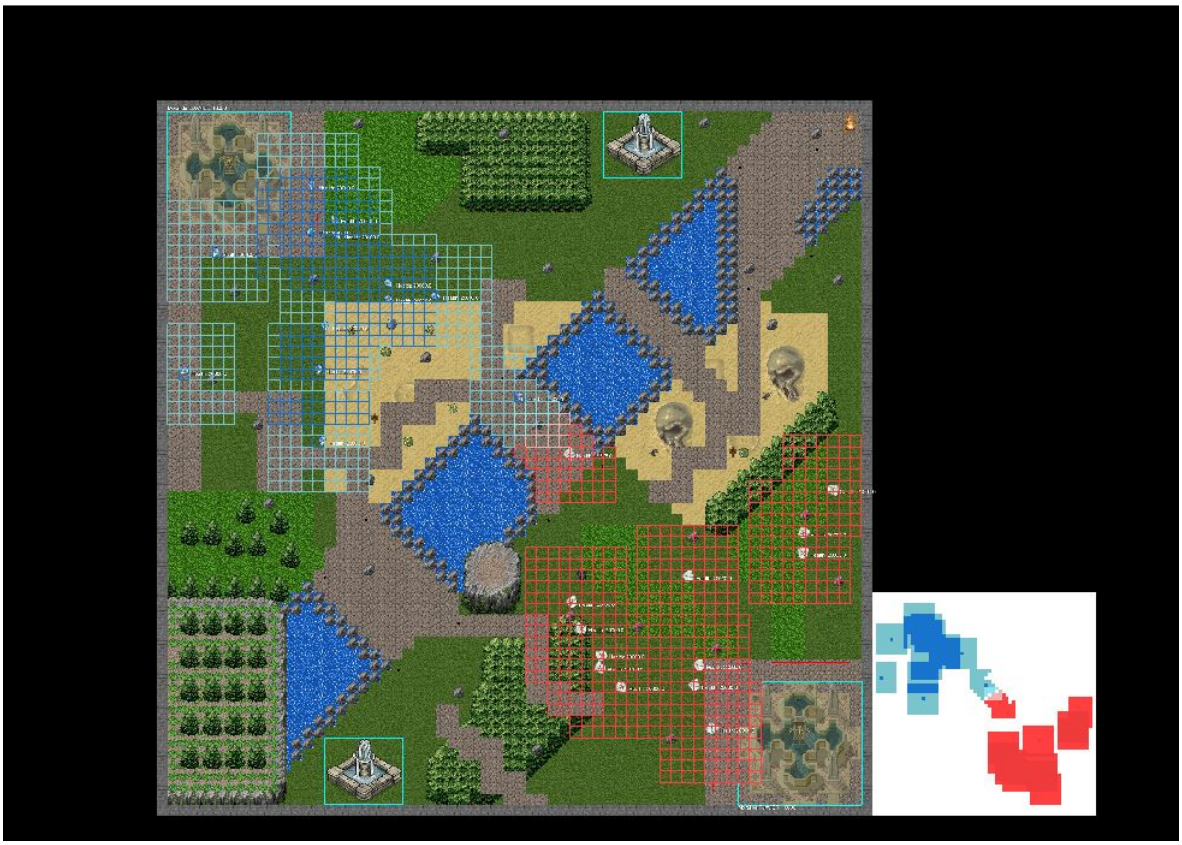

Figura 5: Mapa de influencia.

Parte VII
Flocking

19 Flocking

En esta sección va a tratar sobre la implementación del Flocking pedido en el proyecto. Todo lo referente a esta sección se encuentra dentro del paquete `com.mygdx.iadevproject.aiReactive.flocking`, en la carpeta de test.

Debido a que la finalidad del videojuego no es muy idílica para incluir un flocking en él, hemos decidido implementar el flocking a parte (de ahí que esté en el apartado de test, y no en la parte de código fuente).

Para conseguir este comportamiento, hemos utilizado un árbitro por mezcla ponderada y donde todos los personajes tienen los siguientes comportamientos:

- `CollisionAvoidance` para evitar choques.
- `Separation` para separase del grupo.
- `Cohesion` para juntarse al grupo.
- `VelocityMatching` para ajustarse a la velocidad del personaje que guía al flocking.
- `LookingWhereYouGoing` para que miren por dónde van.
- `Wander` para que parezca que hacen cosas aleatorias.

Cada uno de estos comportamientos tienen un peso distinto, por ejemplo, el `CollisionAvoidance` es el que tiene mayor peso, mientras que el `Wander` es el que tiene menor peso. Para guiar a todo el grupo, se ha creado un personaje que simplemente realiza un `Arrive` a la posición que se clicka en pantalla. Todos los demás personajes hacen el `VelocityMatching` a este personaje.

El funcionamiento del test es el siguiente: hay cinco personajes (cubos) que son a los que se aplica el Flocking y otro personaje (gota) que es al que el grupo de cubos hace el `VelocityMatching`. De esta manera, el usuario puede clickar en la pantalla y la gota lo que hará será ir hacia esa posición realizando un `Arrive`, lo cual hará que los cubos empiecen a ir en esa dirección. Hay que tener en cuenta que la gota **simplemente sirve para modificar la dirección de la velocidad del grupo**; es decir, **el Flocking no se aplica a la gota**.

Parte VIII
Elementos opcionales

20 Elementos opcionales

En esta sección se van a explicar todos los elementos opcionales que también han sido implementados e incluidos en nuestro proyecto.

El primero de ellos ha sido la implementación de distintos comportamientos básico, delegados y en grupo. De hecho, hemos implementado prácticamente casi todos los comportamientos que hemos estudiado y que aparecen en las transparencias de la asignatura. La información sobre todos los comportamientos implementados se podrá encontrar en el apartado correspondiente.

Del mismo modo, también hemos implementado e incluido algunas estructuras de arbitraje como los **árbitros por prioridad** y los **árbitros por mezcla/por pesos** (tanto para los comportamientos acelerados como para los no acelerados). Se podrá encontrar una documentación más detallada en el apartado correspondiente.

Para el caso de las formaciones, se han implementado algunas otras estructuras a parte de la propia formación en circulo. Estos nuevos tipos de formación son: **formación en línea** y **formación en estrella**. Además de esto, también hemos añadido la posibilidad de crear **formaciones de formaciones** todo lo profundas que deseemos (mediante en patrón de diseño *composite*). Esto se explica más detalladamente en el apartado correspondiente.

Uno de los elementos opcionales más importantes que hemos hecho ha sido el **modo depuración**. Durante una partida y pulsando las teclas adecuadas, se puede visualizar un montón de información sobre los movimientos que realizan los personajes, el estado de los personajes o el comportamiento táctico de los mismos. Para ello, en prácticamente todos los comportamientos implementados hemos añadido un método llamado `debug`, que será el encargado de dibujar toda la información de depuración necesaria para ese comportamiento. Además, en la clase `Character` también se han implementado ciertos métodos para mostrar, por ejemplo, el estado táctico en el que se encuentra el personaje. Por otro lado, en las formaciones también se dibujan los puntos a los deben ir los componentes de la propia formación. Para el caso de los waypoints, también contamos con los métodos necesarios para dibujar dichos puntos sobre el mapa del juego.

Tal y como se acaba de comentar, hemos implementado información de depuración para prácticamente todos los comportamientos (entre los que se incluye el Pathfollowing). Por tanto, como este comportamiento se usa para ir a los puntos devueltos por el pathfinding, también se podría decir que ha sido implementado el modo de depuración para el pathfinding, donde se muestran todos aquellos puntos que han sido obtenidos y por donde el personaje deberá ir.

Debido a cómo se han planteado las formaciones, será el propio objeto formación (el ancla) el que ejecute el pathfinding y el que vaya por los puntos que éste devuelva (los componentes de la formación simplemente se limitarán a ir a los puntos que les diga el ancla). Por tanto, para el caso de las formaciones, el pathfinding (normal y táctico) se lleva a cabo en un nivel superior y el camino obtenido se le impone a los componentes de la formación (ya que como acabo de decir, será en ancla el que lo ejecute y no cada uno de los componentes de la formación).

En cuanto a la información táctica implementada, no solo se ha hecho la parte de los mapas de influencia, sino que también se ha añadido un coste táctico a cada uno de los roles implementados en este proyecto (distinto coste para cada tipo de terreno). Ese coste táctico podrá repercutir directamente en la ejecución del pathfindind táctico (siempre y cuando, el personaje que lo ejecuta tenga un rol).

Tal y como comentaba anteriormente, debido a cómo se han planteado las formaciones, será el propio objeto formación (el ancla) el que tome ciertas decisiones y los componentes simplemente se limitarán a ir al punto que se les indique (y a acatar las decisiones tomadas por el ancla). Por tanto, ese mismo principio repercute directamente en las acciones de ataque o curación. Para el caso de las formaciones, la decisión de atacar (y también de curarse) no es tomada por los propios componentes de la formación, sino que se toma en una nivel superior (la toma el objeto tipo formación o ancla y se la impone a sus componentes).

Parte IX
Conclusiones

21 Conclusiones

Este proyecto ha consistido en implementar una serie de elementos de inteligencia artificial que se usan muy comúnmente en los videojuegos reales. Además, la propia implementación de dichos elementos se ha llevado a cabo en el contexto de un juego de guerra en tiempo real. Tal y como se ha podido apreciar a lo largo de la documentación, esos elementos van desde la parte reactiva (los comportamientos más básicos, los movimientos de un punto origen a un punto destino y las estructuras que permiten elegir o combinar el comportamiento o comportamientos a aplicar) hasta la parte táctica (que se ha abordado mediante ciertas estructuras de comportamiento táctico y toma de decisiones, como máquina de estados o árboles de comportamiento), pasando además por otras técnicas interesantes que también se usan hoy día en muchos videojuegos reales (mapas de influencia o waypoints). Así mismo, también se ha diseñado e implementado otra funcionalidad para permitir agrupar a los personajes y gestionarlos de manera compacta (formaciones de personajes).

El estudio, aplicación e implementación de estas técnicas nos ha permitido comprender de una forma más profunda y mejor el funcionamiento interno de los videojuego reales y qué está pasando realmente dentro de ellos cuando vemos determinadas acciones y comportamientos en una partida. Así mismo, también hemos podido reforzar y coger experiencia con algunos conceptos y técnicas muy interesantes que, muy posiblemente, puedan tener una aplicación práctica en otros ámbitos y no solamente en el mundo de los videojuegos. A parte de los anterior, los conceptos estudiados en la asignatura y la realización de este proyecto también nos ha permitido estudiar y aprender la estructura general de un videojuego, cómo éste se organiza y cómo las distintas partes que lo componen interactúan y se comunican entre sí.

Por todo lo anterior, consideramos que estas prácticas han sido muy adecuadas e interesantes, nos han permitido introducirnos (al menos, de forma básica) en el mundo de los videojuegos y en la implementación de éstos y nos han permitido comprender y entender la estructura y organización general de un videojuego.

Parte X
Bibliografía

Bibliografía

[1] Badlogic Games, *LibGDX* Enlace (última visita 16/05/2017).

[2] LibGDX, *State Machine.* Enlace (última visita 15/05/2017).

[3] Profesores de la Asignatura, *Apuntes teóricos.* 2017. AulaVirtual.

[4] Thorbørn Lindeijer et al., *Tiled Map Editor.* Enlace (última visita 06/05/2017).