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 Martín-Blázquez Gómez

jgmarin@um.es

Luis Daniel Hernández Molinero

ldaniel@um.es

Índice

Parte I
Introducción y estructura de la aplicación

1 Introducción

2 Estructura de la aplicación

Parte II
Manual de uso

3 Manual de uso

Parte III
Clase principal y mapa

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 el PathFinding táctico (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.

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.

El control de la interacción del usuario con el juego se ha implementado en la clase `InputProcessorIADeVProject`, 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, y el segundo hace la inversa: transforma una posición del grid a una posición del mapa.

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 1.

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 tres 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.

Figura 1: Mapa diseñado para el proyecto.

- **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`.

Para diseñar el mapa, se ha hecho uso de la herramienta `Tiled Map Editor` [1] 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.

Parte IV
IA reactiva

6 Steerings

7 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.

7.1 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.

7.2 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`.
- `AntiAlign_Accelerated`.
- `Arrive_Accelerated`.
- `Arrive_Accelerated_WithOneRadius`.
- `Flee_Accelerated`.
- `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.

7.3 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 otros objetos. 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.

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

- `Evade`.
- `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 este. 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`.
- `Persue`.
- `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`, 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, aquél 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.

  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.

7.4 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`.
- `Separation`.

8 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 6, 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 y la máxima rotación (en el caso del acelerado), y la máxima velocidad y máxima orientación (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.

9 Modelo

Parte V
PathFinding

10 PathFinding

Parte VI
IA táctica

Parte VII
Conclusiones

11 Conclusiones

Parte VIII
Bibliografía

Bibliografía

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