Desarrollo de Videojuegos con Inteligencia Artificial

Alejandro García-Muñoz Muñoz y Adrián Salgado Jimeno

Curso 2020/2021. Fecha: 24 de octubre de 2020.

1.- Introducción

1.1- Contexto

La Inteligencia Artificial (IA) es una disciplina de las ciencias de la computación que busca la creación de programas y autómatas con la capacidad de interactuar con su entorno, demostrando un grado mínimo de inteligencia. En suma, lo que se busca es que estos sistemas actúen de forma racional.

El origen de la IA puede remontarse a la Prueba de Turing, una especie de examen ideado por Alan Turing en 1950 que consiste en que un interlocutor (que puede ser una máquina o un ser humano) y un evaluador humano mantienen una conversación. Dicho examen es superado si después de la conversación, el interlocutor (en caso de ser una máquina) es indistinguible de un ser humano por parte del evaluador.

1.2- Motivaciones

En el contexto del Desarrollo de Videojuegos, el mercado de videojuegos goza de una amplísima variedad de títulos en los que se implementa la inteligencia artificial como un recurso más de estos. Por ejemplo, en juegos como Minecraft, los zombis al perseguir al jugador buscan el camino más corto hacia este y (en las versiones más recientes) supera los desperfectos del terreno (agujeros, montañas, árboles...). Por tanto, es imprescindible para un buen programador conocer los fundamentos básicos de la IA para poder realizar una buena implementación de esta.

Otro ejemplo bastante icónico del uso de la IA son todos los juegos de *Agom* (según la clasificación del sociólogo Roger Caillois), en los que el elemento predominante del juego es la competencia contra otro jugador. Siendo más concisos, el papel de la IA en este escenario es sustituir a un humano que no pueda estar disponible para jugar.

1.3- Objetivo

El objetivo a conseguir en esta práctica es conseguir la implementación de un algoritmo de búsqueda informada con heurística débil que permita a un personaje controlado por IA encontrar la salida de un terreno basado en una cuadrícula de celdas.

El objetivo, por tanto, es que dicho personaje sea capaz de analizar el terreno y encontrar una salida en tiempo de ejecución sin conocer de antemano el terreno. Así, este terreno se genera de forma aleatoria antes de que nuestro personaje pueda comenzar a procesar un plan de salida del terreno.

2.- Descripción del entorno

2.1- Entorno de desarrollo

Esta práctica se ha llevado a cabo en el motor de videojuegos Unity, descargando para esta práctica el código encargado de la lógica del videojuego previa a la adición de la IA. De esta forma, el proyecto en su etapa inicial ya consta de la lógica encargada de generar el terreno aleatorio y de implementar un nuevo comportamiento al personaje controlado por IA.

2.2- Elementos del escenario

Una vez cargada la escena, podemos distinguir en ella los siguientes elementos:

1- Un escenario creado a base de celdas cuadradas, todas ellas con coste de movimiento uniforme, con valor $g = 1$.
2- El personaje controlado por IA.
3- Muros que ocupan el espacio de una celda e impiden que esta sea transitable. Tienen un coste asumido de $g = \infty$.
4- El objetivo a alcanzar por el personaje.
5- Enemigos que se desplazan de forma aleatoria por el terreno.

2.3- Reglas del escenario

Además, durante la ejecución de la práctica se dispone de la siguiente información y de una serie de reglas fijas:

1- El tamaño del escenario (número de celdas de largo y ancho) será siempre conocido en tiempo de ejecución. Al iniciar la escena.
2- Los obstáculos son fijos dentro del escenario y el mapa no se varía en ningún momento de la ejecución.
3- El agente no puede estar en una celda ocupada por un muro.
4- Se permite el movimiento en las cuatro direcciones cardinales N-S-E-O.
5- Para superar la práctica el personaje, este debe colocarse en la celda donde se encuentra la meta.

Los muros son generados de forma aleatoria estableciendo una semilla inicial, especificada en el objeto *Loader*, encargado de la lógica de creación del escenario.

3.- Desarrollo de un controlador de IA

3.1- El componente `AbstractPathMind`

Para desarrollar el controlador por IA que se implementará al personaje, se debe añadir un componente de tipo Script que herede de la clase `AbstractPathMind`. Esta clase tiene un método virtual que permite obtener la siguiente dirección que debe seguir el personaje. Esta clase nos sirve como interfaz entre las diferentes implementaciones de los algoritmos de búsqueda de un camino.

La clase que viene por defecto implementada en el comportamiento del personaje era `RandomMind`, que al heredar el método que obtiene el siguiente movimiento, este solamente devuelve un movimiento aleatorio. De esta forma, es muy complicado que encuentre el camino correcto.

3.2- Búsqueda de un camino de salida

Por tanto, para implementar un algoritmo de búsqueda de un camino de salida, se debe utilizar una nueva clase que implemente un algoritmo capaz de encontrar el camino hasta la salida.

El algoritmo que se ha decidido implementar es el algoritmo A*. Este algoritmo combina el algoritmo de búsqueda con coste uniforme de Dijkstra con una función heurística. Esta heurística consiste en una intuición sobre qué casillas son más prometedoras, empleando para esto la distancia Manhattan.

La distancia Manhattan se obtiene mediante la suma de la distancia en filas y la distancia en columnas entre la casilla de salida y la casilla de llegada; por tanto, esta heurística es la más indicada en nuestro caso ya que los movimientos solamente están permitidos en las cuatro direcciones cardinales. Toda esta información se guarda en una estructura de datos de tipo Nodo.

Para poder implementar este algoritmo, hace falta crear una lista abierta (aunque también funciona si se emplea una cola) en la que se vayan metiendo los distintos nodos, cada uno de los cuales tiene a su vez un nodo como padre, que es lo que permite es ir creando un camino de salida. Este camino se puede ir almacenando en una pila que, una vez obtenido, se puede ir sacando movimiento a movimiento para que lo siga.

4.- Características de diseño e implementación

4.1- Clase `AStarMind`

Como se ha indicado anteriormente, la clase `AStarMind` debe heredar de la clase abstracta `AbstractPathMind` para poder sobrescribir el método encargado de obtener la siguiente dirección a seguir. Esta clase es la encargada de declarar una pila en la que almacenar la lista de direcciones que se deben seguir y de aplicar el algoritmo A* para encontrar un camino.

Esta implementación del algoritmo A* devuelve únicamente el último nodo visitado de la lista, que hace referencia de forma recursiva al resto del camino a seguir. Esto lo que permite es añadir los nodos en orden inverso a la pila, para que a la hora de extraer los nodos de la pila (al tratarse de una estructura lineal LIFO), estos se extraigan en el orden correcto.

En caso de no encontrar un camino o de quedarse bloqueado por muros (véase el caso de la semilla 4312), dicho algoritmo retorna un valor nulo. Por tanto, en este caso se indica por consola al usuario de que nuestro personaje no ha sido capaz de encontrar un camino hasta la salida.

4.2- Clase `Node<T>`

La clase `Node<T>` es la encargada de gestionar la estructura de datos de tipo Nodo, la cual almacena el coste de dicho nodo (esto es importante a la hora de decidir cómo se ordenan los nodos en la lista abierta), la dirección en la que te ha llevado dicho nodo y la distancia Manhattan de dicho nodo (como función heurística).

Se ha decidido hacer una implementación genérica de la estructura de datos Nodo, ya que así se aumenta la reusabilidad del código y solamente se debe cambiar ligeramente la implementación de algunos detalles del algoritmo A*.

En cuanto a los métodos que implementa `Node<T>`, podemos comprobar si dos nodos son iguales (con esto se permite finalizar el algoritmo A*), expandir el nodo y comparar dos nodos (para poder ordenarlos después dentro de la lista abierta). Al expandir el nodo, es importante descartar valores nulos, ya que estos pueden ocasionar problemas al insertarlos en la lista abierta, pues deben ser comparados y expandidos.

Por otro lado, la clase `Node<T>` implementa a su vez la interfaz del lenguaje C# `IComparable<T>`. Esto permite que la lista abierta ejecute el método implementado dentro de la estructura de datos Nodo. Como criterio de desempate, en este contexto se ha decidido que si dos celdas (y, por extensión, nodos) tienen la misma distancia Manhattan, decida quedarse con aquella que mueva el personaje a la derecha.

5.- Discusión sobre los resultados obtenidos

Una vez implementado el algoritmo en sus respectivas clases, al ejecutarlo se observa que el personaje encuentra el camino más corto y lo sigue. En caso de empate, decide ir hacia la derecha todo lo que pueda hasta que se encuentre con un obstáculo o tenga que cambiar de dirección. Según la situación, en el primer movimiento en ocasiones decide ir hacia arriba, porque en el primer movimiento no se llega a aplicar el criterio de desempate.