IA-P2: El Hilo de Ariadna

1. Preparación

En esta práctica continuamos el mismo proyecto que con la práctica 1, reorganizando las carpetas para que pueda escalar bien, y añadiendo todos los materiales ofrecidos en el campus para poder hacer esta parte. Como siempre, el repositorio del proyecto se encuentra en GitHub: *https://github.com/jeramauni/IA_2020*.

2. Desarrollo

a. Entorno virtual

El **laberinto** es una cuadrícula de **n*m** (Ambas dimensiones modificables desde el editor en la clase *GameManager*), en la que todas las casillas son visitables. Los muros son un complemento de cada casilla, y esta se encarga de saber dónde tiene un muro para así poder comunicarlo a los agentes que se desplacen por ella. En cada ejecución se elabora un laberinto diferente, generado procedimentalmente de manera aleatoria mediante el algoritmo de *Hunt&Kill.*

Este sistema de generación está *complementado* con la posibilidad de seleccionar entre generación de *laberinto* ***perfecto*** (un solo camino recorre el laberinto desde el punto de salida a la esquina opuesta) o ***imperfecto*** (se encuentran caminos circulares por lo que pueden haber varias opciones para llegar a una misma casilla) mediante un flag en la entidad *GameManager*.

La casilla *(0, 0)*, abajo a la izquierda, es siempre la casilla de salida. Hemos resaltado esta condición pintándola de un color llamativo. En la casilla de salida se ubica el

avatar, **Teseo**, al inicio de la partida. El movimiento por cursores (cuando la barra espaciadora no está pulsada) de Teseo ha sido implementado de la siguiente manera: el jugador solo puede moverse por carriles de un tamaño de *0,2 uds*. de Unity. Cada casilla tiene una ***cruz virtual*** que representa los carriles por los que puede moverse el jugador. Si hay un muro en una dirección, entonces el brazo de la cruceta en dirección a ese muro se capa, de tal forma que el jugador no pueda ir hacia allí. Esto puede generar lo que a simple vista podría parecer un error por el cual en los espacios abiertos el jugador solo se puede mover por carriles, pero de ninguna manera lo es, pues está implementado así para respetar los movimientos al estilo cuadrícula.

En el centro del laberinto, por contraparte, está el **minotauro**. El movimiento del minotauro está implementado mediante un *merodeo aleatorio* en el que al inicio se ***elige una dirección*** en la que avanzar siempre que no haya una pared delante, y cada vez que se encuentra con una pared en su avance debe cambiar su rumbo hacia una nueva dirección válida. Al ser aleatorio hemos considerado que el minotauro pueda darse la vuelta al llegar a una esquina, ya que en cierto modo es la gracia de que sea aleatorio... que en cualquier momento pueda girarse hacia el jugador.

También es importante resaltar que hemos creado un script para el ***movimiento de la cámara***, la cual al inicio se mueve *suavemente* desde el centro hasta el jugador, y después con la misma suavidad lo va siguiendo según su movimiento.

b. Suavizado

El suavizado supone un reto pues realmente hay pocos sitios en los que por un pasillo se pueda realizar. En nuestro caso, hemos propuesto un suavizado que consiste en ***avanzar diagonalmente*** en el caso que se pueda. Así, por ejemplo, en el caso de que estuviéramos en una casilla que no tuviera muro izquierdo ni inferior, entonces podría moverse en esa diagonal. Esta funcionalidad está siempre activa, ya que, combinada con la heurística, hacen que la elección de caminos siempre sea la más intuitiva y lógica.

c. Hilo de Ariadna

Para representar el camino que debe recorrer Teseo, hemos optado por representar con bolas todos los nodos que explora, diferenciándolos según tres tipos:

1. Bolas grandes blancas: Representan el camino real de Teseo, o sea, el camino más corto según *A**.
2. Bolas medianas marrones: Representan todos los elementos de la lista cerrada que no llegaron a formar parte del camino, pero que llegaron a ser una opción real.
3. Bolas pequeñas azules: Representan todos los elementos de la lista abierta, los cuales nunca llegaron más allá de este tipo pues tenían costes mayores.

Como el minotauro puede en cualquier momento ponerse en mitad del camino del jugador, el camino ***vuelve a ser calculado*** cada vez que Teseo cambia de casilla para que no use caminos obsoletos.

Para ofrecer más información, hemos implementado una sistema visual para el jugador. Se trata de una ***interfaz*** *sencilla* que muestra en la pantalla un texto donde

aparecen **métricas** sobre el algoritmo *A**, el **tiempo** que le ha llevado calcular el camino, está expresado en milisegundos. Como el algoritmo se ejecuta cada vez que se cambia de casilla el tiempo de cálculo también se actualiza en función de este nuevo cálculo, también aparece el número de nodos explorados en cada ejecución del *A**.

d. Movimiento Automático de Teseo

El movimiento de Teseo está dividido en dos scripts: *AutoMov.cs* y *TeseoMov.cs*.

El script *AutoMov.cs* de Teseo hace que vuelva a la casilla de inicio mientras se mantiene pulsada la tecla ***barra espaciadora***. Este movimiento se realiza mediante interpolación lineal entre posiciones de baldosas de una lista de baldosas calculada por el *A** aportado por *TeseoMov.cs*.

Para hacer *A** decidimos obviar todo el material que teníamos y aprender de cero como funciona ***matemáticamente*** el algoritmo, para así tener claro cada uno de los pasos que debe ejecutar. De esta forma, tenemos un método en el script *TeseoMov.cs* que se encarga de hacer todo el camino en unos sencillos pasos en bucle dada una *celda inicial*:

1. Coger los vecinos a esa casilla, descartando las casillas adyacentes en las que hay un muro.
2. De los vecinos, quedarnos con la celda más cercana.
3. Si desde la celda más cercana, si alguno de los vecinos anteriormente calculados tiene un camino más corto, actualizar ese camino.
4. Sumar al coste total el coste hasta la celda más cercana.
5. Ahora la celda actual es la celda más cercana, y repetimos.

Hay una variable determinante en todo este camino y es la ***heurística*** usada. La heurística que hemos implementado nosotros es una mezcla de *Manhattan* con la diagonal de un cuadrilátero regular. De esta forma, el coste de cada movimiento horizontal o vertical es un valor a nuestra elección, pongamos 10. Sin embargo, si el movimiento tiene que ser diagonal, entonces el coste de cada movimiento es la hipotenusa del ***triángulo rectángulo isósceles*** cuyos catetos valen el valor de antes, 10.

Esto quiere decir que si buscamos la posición *(0, 0)* y estamos en la *(2, 1)*, el coste total del movimiento será 24,14:

$$10,00 \; de\ ir\ de \; (2,1) \; a \; (1,1)$$
$$14,14 \; de\ ir\ de \; (1,1) \; a \; (0,0)$$

Esto surge de dada la posición *origen (x, y)* y la *destino (0, 0)*, con la fórmula:

$$costeHeurístico = |x - y| + \sqrt{|x - y|^2 + |x - y|^2}$$

Cuyos sumandos corresponden a una recta y una hipotenusa de catetos de valor idéntico al primer sumando.

3. Ampliaciones

a. Generación procedimental del laberinto

La base de toda la generación del laberinto es la clase *MazeCell*, 'la baldosa' que constituye el laberinto. Esta clase contiene tanto la posición de las baldosas como información sobre si tiene o no paredes a su alrededor. Con esta premisa en mente, lo primero que se hace al iniciar el juego es crear un laberinto del tamaño indicado y con todas las paredes colocadas. A partir de este momento se ejecuta el algoritmo de *Hunt&Kill* que consiste en:

1. Se elige una ubicación inicial.
2. Se realiza una caminata aleatoria por el laberinto, tumbando paredes hacia vecinos no visitados hasta que se llegue a una celda sin vecinos sin visitar.
3. Se entra en modo 'Hunt' donde se busca una celda no visitada adyacente a una visitada.
4. Se repite la 2 y 3 hasta que 'Hunt' no encuentre celdas sin visitar.

Durante este proceso se actualiza el mapa de baldosas actualizando el estado de las paredes de las baldosas afectadas.

En último lugar si previo a la ejecución se selecciona que se desea un laberinto imperfecto, un último algoritmo (Imperfections) entra en funcionamiento una vez finaliza HuntAndKill.

Este algoritmo destruye paredes aleatorias del interior del laberinto para crear más de un camino posible para llegar al final. El número de paredes destruidas aumenta proporcionalmente a las dimensiones del laberinto siguiendo la función:
número de paredes = **n** * **m** / 14.

b. Camino concreto del minotauro

Esta funcionalidad, a pesar de que no ha sido añadida, debemos comentarla pues dado que todos los laberintos que generamos son ***procedimentales***, es *imposible* hacer que el minotauro siga un camino fijo, más allá de que podamos hacer que vaya de una casilla a otra por el camino más corto que encuentre usando el ya implementado *A**.

4. Pruebas

En cuanto a las pruebas nuestro entorno está completamente parametrizado para poder regular el tamaño del laberinto y crearlo de diferentes formas. Todos estos parámetros están en el script del cargador de laberinto (*MazeLoader.cs*) contenido en el objeto *GameManager*. Esto nos ha permitido hacer distintas pruebas con diferentes laberintos para ver que funciona correctamente.

Todas las pruebas que hemos realizado han sido durante el propio desarrollo de la práctica, mediante las herramientas de depuración que el propio **Unity** nos proporciona, y todas ellas en la misma escena.

Para cuando se hagan pruebas, es imprescindible no desactivar scripts, dado que la comunicación entre los personajes es clave para el buen funcionamiento de la práctica. Dicho esto, entre otras cosas, se puede *modificar el* ***tamaño*** del laberinto tanto en su anchura como en su altura. También se puede *modificar la* ***velocidad*** de Teseo, tanto en su movimiento normal, como en el automático. Otra cosa que se puede modificar es el *coste del* ***movimiento***, yendo el coste normal de movimiento desde *0* hasta *10*, y el coste del movimiento de la casilla sobre la que está el minotauro de *10* a *50*. Por último resaltar que, al igual que la velocidad de Teseo, también se puede modificar la del **minotauro**.

A la hora de comprobar si el algoritmo hace bien el **esquivar** al minotauro, lo mejor es, obviamente, acercarse a él. Cuando pasemos cerca de él, el algoritmo, que seguía un camino uniforme, *buscará una ruta alternativa*, la cual seguramente haga recorrer al Teseo un camino *mucho más largo*. No es de extrañar que, como este camino se calcula cada vez que teseo avanza, y para ese entonces el minotauro ya habrá cambiado de casilla, que ***cambie de camino varias veces cuando el minotauro está rondando cerca***. Esto es totalmente normal, por lo ya comentado.

También hay que comentar otra característica, y es que Teseo ***no está programado para huir del minotauro***, solo para escoger el camino más corto. Y si el camino más corto, ya sea por que no hay otro o porque de todos los que hay se ha elegido ese, pasa por atravesar por la casilla del minotauro, Teseo lo hará sin vacilar.

Dado que el minotauro *siempre va hacia delante* hasta que encuentra un muro, esto nos puede facilitar mucho el comprobar si el algoritmo funciona correctamente, pues el minotauro puede fácilmente encontrar un bucle en un pasillo, y podemos aprovechar eso para ***ejecutar el movimiento automático*** cerca de ese pasillo.

Un recordatorio: si al lanzar el ejecutable de la práctica le das a ALT+ENTER, se pone la pantalla completa y facilita la visión del *HUD*.