PRÁCTICA 2

LOS EXTRAÑOS MUNDOS DE BELKAN

Antonio Jiménez Rodríguez 2D Subgrupo D2

1. Modificaciones de jugador.hpp

La primera modificación a observar es ***struct nodo*** donde se han añadido los booleanos zapatillas y bikini. Estas variables no serán necesarias para la búsqueda en profundidad y búsqueda en anchura pero si para el costo uniforme. Un nodo a explorar es distinto de otro, además de por tener fila, columna u orientación diferente, por tener o no zapatillas y por tener o no bikini.

Con respecto a la clase ***ComportamientoJugador*** hemos añadido las siguientes variables:

/* Estas seis variables sólo se utilizan para el nivel 2 */

- int tiempo_restante: el tiempo que le queda al nivel para terminar la simulación.
- estado ubicacion_carga: donde guardaremos el valor de la columna de la casilla para cargar la batería.
- bool cargar: utilizamos la variable para saber si vamos a cargar la batería.
- int objetivo_actual_f: guardamos la fila del objetivo actual.
- int objetivo_actual_c: guardamos la columna del objetivo actual.
- bool equipando_zapatillas: utilizamos la variable para saber si hemos detectado unas zapatillas con el sensor.terreno y aun no tenemos las zapatillas.
- bool equipando_bikini: utilizamos la variable para saber si hemos detectado un bikini con el sensor.terreno y aun no tenemos el bikini.

/* Estas dos variables sólo se utilizan para el nivel 1c y nivel 2 */

- bool zapatillas: utilizamos la variable para saber si tenemos zapatillas o no.
- bool bikini: utilizamos la variable para saber si tenemos bikini o no.

/* Esta variable se usará en todos los niveles */

- hayplan: variable para saber si se ha encontrado plan hacia el objetivo.

2. Modificación de jugadores.cpp

1. Structs

Tenemos 3 tipos de structs diferentes, **nodo**, **nodoCosto**, **nodoEstrella.**

El **struct nodo** lo utilizo para la búsqueda en profundidad y en anchura. Este solo tiene una variable estado que representa los valores del nodo y la lista de acciones que habría que ejecutar para llegar hasta a él desde el nodo en el que estemos.

El **struct nodo** lo utilizo para la búsqueda de costo uniforme (Nivel 1.c). Lo que diferencia a este struct del anterior es la variable costo utilizada para guardar el coste que tiene ir a ese nodo desde donde estamos, y el operador < utilizado para la cola de prioridad.

El **struct nodoEstrella** lo uso para el Nivel 2. Las componentes de este son las siguientes: el estado del propio nodo lo conserva pero en vez de una lista de acciones se guarda la acción que hay que hacer para llegar desde su padre a él. Además de esta información también almacena el estado del padre del nodo, la **g** y la **h** utilizados para calcular el costo del nodo.

2. Método think

En este se pueden diferenciar varios bloques, la carga, coger zapatillas o bikini, pintar el mapa, y construir los planes para el desplazamiento de nuestro personaje.

Lo primero que hacemos en el método think es ir pintando el mapa para el Nivel 2 llamando a **pintarMapa(...)**. En esta función utilizamos un switch para tener en cuenta la orientación de nuestro jugador y pintamos lo que vemos de mapa con el vector **sensores.terreno**.

Después comprobamos si tenemos que rehacer el plan de carga o el que estábamos siguiendo al objetivo. Nuestro método rehace un plan cuando nos colocamos delante de una casilla que sea agua o bosque (las de mayor coste) y en cualquier casilla del **sensor.terreno** había una casilla que no habíamos explorado previamente. Para esto usamos la función auxiliar **casillaDesconocida(...)**. Para saber si era desconocida antes de hacer el plan hacemos una “fotografía” del mapa antes de cada llamada a **pathFinding_Nivel2(...)**.

Ahora comprobamos si es necesario ir a cargar. La heurística implementada aquí es la siguiente: en el momento que la batería esté por debajo de 800, no tengamos suficiente batería para ir del punto actual al nodo destino y del nodo destino a la casilla de carga, y el número de movimientos para realizar esto sea menor al tiempo restante del programa deberemos ir a cargar la batería. Todo esto está implementado en la función **Cargar(...)**. Además en esta función se llama a la

función **asignarCasilla(...)** donde se comprueba que casilla de carga es la que más nos conviene para ir. Para eso utilizo otra función auxiliar **calcularRecarga(...)** que tiene el cuerpo del algoritmo A* pero me devuelve los datos del costo del camino y del tamaño del plan, no modifica el plan actual ni la variable **hayplan**.

Por otra parte, el plan también se rehará si hemos llegado al destino o si el personaje ha descubierto un precipicio o muro y va a caerse o chocar.

Después de esto calculamos el plan distinguiendo si es plan de cargar batería o es plan para ir a por el objetivo. Pero esta acción puede quedar a un lado si el sensor del terreno encuentra un bikini o unas zapatillas y no tenemos equipado el correspondiente objeto. En este caso se creará un plan que irá directamente hacia el objeto para equiparselo y continuar con el plan anterior una vez recoja las zapatillas o bikini. Este bloque solo se ejecutará hasta que el personaje consiga las dos prendas.

Por último es momento de ejecutar el plan. Si hay plan se ejecutará la acción correspondiente. En el caso de estar en el Nivel 2 si nos encontramos con un aldeano he decidido esperar a que se aparte ya que no obtenía una mejoría notable al replanificar para esquivarlo. En el caso de que no haya plan tenemos que ver en qué caso estamos. Si estamos en la casilla de carga nos quedaremos en ella hasta que se cargue la batería necesaria para continuar. Y si estamos en la casilla de bikini o zapatillas pondremos las variables booleanas utilizadas a false y así poder continuar en busca de objetivos.

3. Método pathFinding_CostoUniforme

En este método comentar que tenemos el set de explorados, y una cola de prioridad de nodoCosto.

Lo primero que hacemos ahora en el bucle es comprobar si estamos sobre zapatillas o bikini. Esto también lo hacíamos en el método think pero no estamos hablando de la misma variable, ya que al generar las distintas ramas de nodos alguna rama conseguirá los accesorios y otras no por lo que tenemos que diferenciar.

Posteriormente calculamos el coste de la acción sobre la casilla en la que estamos situados y la generación de los hijos del nodo.

4. Método pathFinding_Nivel2

En este método he decidido implementar el algoritmo A* ya que es el más rápido para la búsqueda de costo uniforme. Para este método utilizamos el **struct nodoEstrella** cuyas diferencias con el **struct nodo** han sido explicadas anteriormente, una cola con prioridad y los nodos explorados además de guardar el estado los guardamos íntegros en una lista.

La ejecución del algoritmo sería la siguiente: partimos del nodo origen inicializado con el estado del propio nodo, el **nodo padre** (que en este caso no tendría asique le asignamos los valores -1), la **g** (con valor 0 ya que estamos en el nodo origen y la **h** que es la distancia Manhattan hasta el nodo destino.

Empezamos el bucle while comprobando si el nodo está o no en explorados:

- Si no está en explorados:

Lo primero de todo, como vamos a generar los hijos, guardamos en en el estado del padre al nodo actual que estamos explorando. Ahora comprobamos si estamos sobre el bikini o las zapatillas, en caso de que si actualizamos la variable correspondiente del nodo hijo a verdadero. Después de esto toca el cálculo del coste, que fácilmente calculamos con unos **if else** anidados para qué depende de sobre que tipo de casilla estemos, y si tenemos o no bikini/zapatillas, el coste de la acción a realizar sea uno u otro. El coste de las acciones se va acumulando en cada nodo hijo.

Estas operaciones podemos calcularlas antes de generar los hijos ya que para los trés los valores van a ser los mismos.

Ahora pasamos a la generación de los hijos del nodo.

Lo primero que hacemos es calcular la **h** de los hijos girar a la izquierda/derecha. Calculamos también su orientación y lo añadimos a la cola con prioridad de explorar si aún no ha sido explorado. Si ha sido explorado habría que comprobar el nodo hijo como al principio del bucle (esto será explicado más adelante).

Una vez añadidos los hijos correspondientes a los giros comprobamos si el hijo que avanza se puede crear (si tiene o no obstáculo delante). Si puede, se calcula su **h** y lo añadimos a la cola con prioridad de explorar si aún no ha sido explorado.

- Si está en explorados:

Llamamos a la función **actualizarExplorados(...)**. Lo que hacemos en esta función es comprobar si la **g** del nodo pasado por parámetros es menor que la **g** del nodo que está en explorados con el mismo estado. Si es así, significa que existe un camino mejor para llegar a ese nodo, por lo que tendríamos que sacarlo de explorados y meter de nuevo en explorar al nodo con menor **g**. Esto se hace para “actualizar” a los hijos que había generado previamente el nodo.

El bucle while terminará cuando lleguemos al nodo destino o nos quedemos sin nodos para explorar.

Finalmente en este método para devolver el plan habría que recorrer los ancestros del nodo destino para obtener las acciones del plan, de ahí que guardaramos los nodos explorados en una lista.

En el cálculo del coste del plan tenemos la casilla ‘?’.

He decidido ponerle un coste a esta casilla de 2. Lo he hecho porque he realizado ejecuciones en los diferentes mapas y este valor es el que mejor resultado me ha dado.