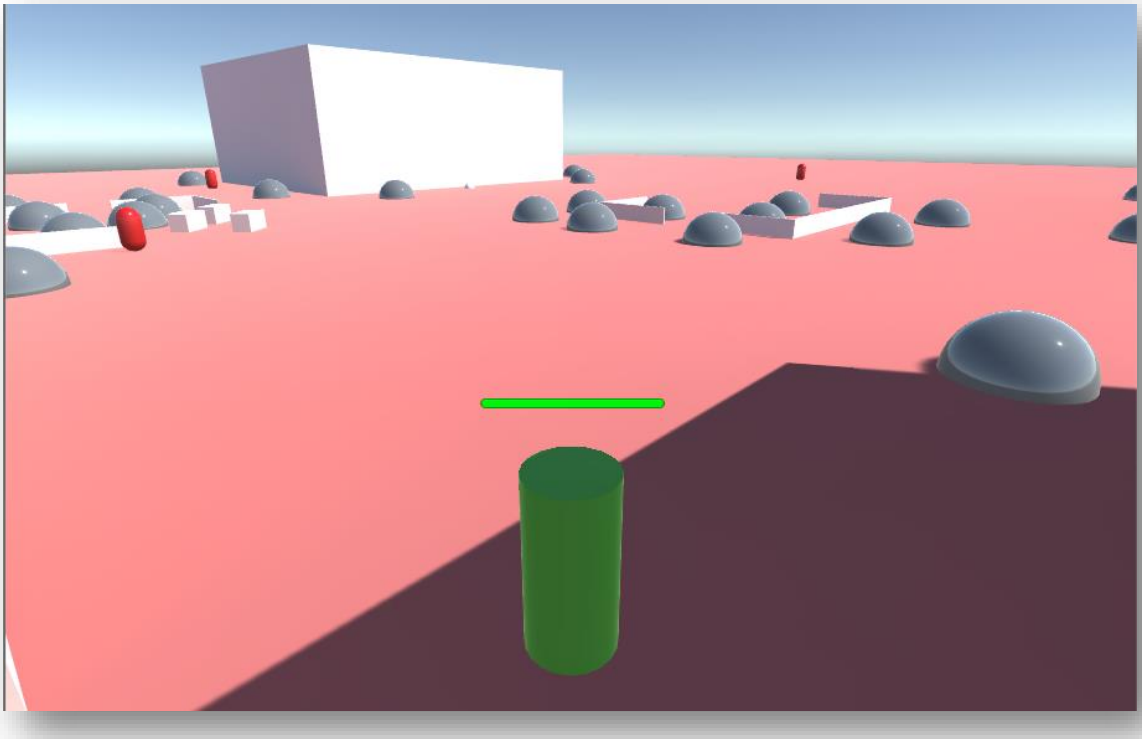
PROYECTO PRÁCTICO WESTERN AI. ENTREGA FINAL.

VÍCTOR GARCÍA CORTÉS



3 DE JUNIO DE 2019

DISEÑO Y CREACIÓN DE VIDEOJUEGOS. ESNE

GRUPO: 3.3

Índice.

Características del diseño e implementación.

Para la realización de esta entrega final partí del segundo entregable en el cual hice una primera implementación de cómo iba a ser el funcionamiento del proyecto. Ampliando en todos los aspectos las funcionalidades que tenían los componentes y por último implementar un algoritmo, en mi caso fue A*. En el segundo entregable fue cuando implementé una primera fase del funcionamiento de los enemigos. Esto tenían 4 estados (patrol, wander, attack y runaway), esto lo implementé por medio de un controlador stateAI (Anexo 1.) en el cual iba comprobando varios parámetros los cuales cambiaban su comportamiento:

- La posición del enemigo y la del player.
- La distancia al player.
- Rango de visión del enemigo.
- Distancia de ataque.
- Vida del player.

Dependiendo de estos estados su comportamiento era uno u otro.

El primer paso fue que los enemigos detectaran al player y se movieran hacia él, sin ningún tipo de condición, es decir ellos sabían en todo momento donde estaba el player y se movían hacia él. A continuación implementé varias parámetros y funciones a los enemigos. Mediante una tabla de estados (Anexo 1.) iba a darles un comportamiento u otro. Para cambiar el estado de los mismos añadí un rango de visión, que iba a permitir detectar al player si este está dentro del rango de visión, a continuación, añadí una distancia de ataque, estos se pararán a esta distancia y dispararán al player para intentar acabar con él.

Esto lo conseguí lanzando raycast desde la posición del enemigo en su Vector.forward y detectando mediante su tag si era el player o no. En este momento no estaba implementado el estado patrol por lo que el estado por defecto que tenían los enemigos era wander ya que sabía en todo momento donde está el player. Por tanto su estado cambiaba de wander a attack si la distancia entre el player y el enemigo era menor que una variable constante llamado minDistance. Al mismo tiempo para dar movimiento a los enemigos hice uso de la función **MoveTo()** ya creada en el script **CowboyLocomotion.cs (**Anexo 6.), que permitía mover a un agente hasta una posición dada, esta la utilice para toda funcionalidad que requiriera de movimieto.

Hasta el momento los enemigos se mueven hasta el player y se paran cuando están a distancia de ataque, pero no hacen nada más porque aún no había implentado la mecánica de ataque. Esto lo implementé una vez que funcionaba la máquina de estados.

Pasamos al estado patrol, partiendo del script CoverSearch.cs (Anexo 3.) donde tenemos una lista de GameObjects que están todas las posibles covers que hay en escena. Este script nos va a permitir recorrer todas esas covers y hacer que los enemigos se muevan a una de ellas en caso de que el player no esté en su rango de visión. Cogiendo la referencia de todas estas covers que hay en escena, al comenzar la partida se le asigna una de cover random a cada enemigo, esto hace que los enemigos se muevan a una diferente cada vez ya que, por el contrario, todos irían a la misma cover y seguirían el mismo orden, no queremos esto.

Por tanto al implementar esto los enemigos tienen uno comportamiento, más o menos, aceptable. Comienzan en estado patrol, una cobertura es asignada aleatoriamente y se mueven

a ella, en caso de que el player entre en su ángulo de visión su estado pasará a wander, si el player sale de su ángulo de visión volverán a patrol para seguir buscando a player, si este no sale de su ángulo de visión y llega a estar a distancia de ataque, el estado del enemigo pasará a attacking.

Anexo 11.

Por el momento no hay mucha dificultad pero tenemos unos enemigos que tienen diferentes estados con diferentes objetivos. Partiendo de este punto, toca implementar la funcionalidad al estado attacking, que los enemigos puedan hacer daño al player.

Para llevar esto a cabo, tuve que añadir un nuevo script que será **Health.cs (**Anexo 7**.)** donde le voy dar vida al player. En este script he creado:

- Variables:
  - Una variable para la vida actual del player
  - Vida máxima que este puede tener.
- Métodos:
  - **TakeDamage**(), este método recibe un parámetro de daño (cantidad de daño recibido y la probabilidad de daño).
  - **UpdateHealth**(), aumentará la vida de player a lo largo del tiempo

Y para su representación visual:

- Un slider para representar la vida.
- Un sistema de partículas y sonido para indicar que has sido dañado.

Cuando la vida llega del player llega a 0 desactivamos el gameObject y su heathBar, de esta manera indicamos que el player ha muerto.

Esto es algo sencillo de implementar pero que viene muy bien a la hora de ver visualmente si estás recibiendo daño, la cantidad de vida que tienes y que te queda y sobre todo darle algo de jugabilidad…

La segunda parte es añadir a los enemigos un script de disparo para que puedan quitar vida al player. Este script lo he llamado **CowboyAttack.cs (**Anexo 5.).

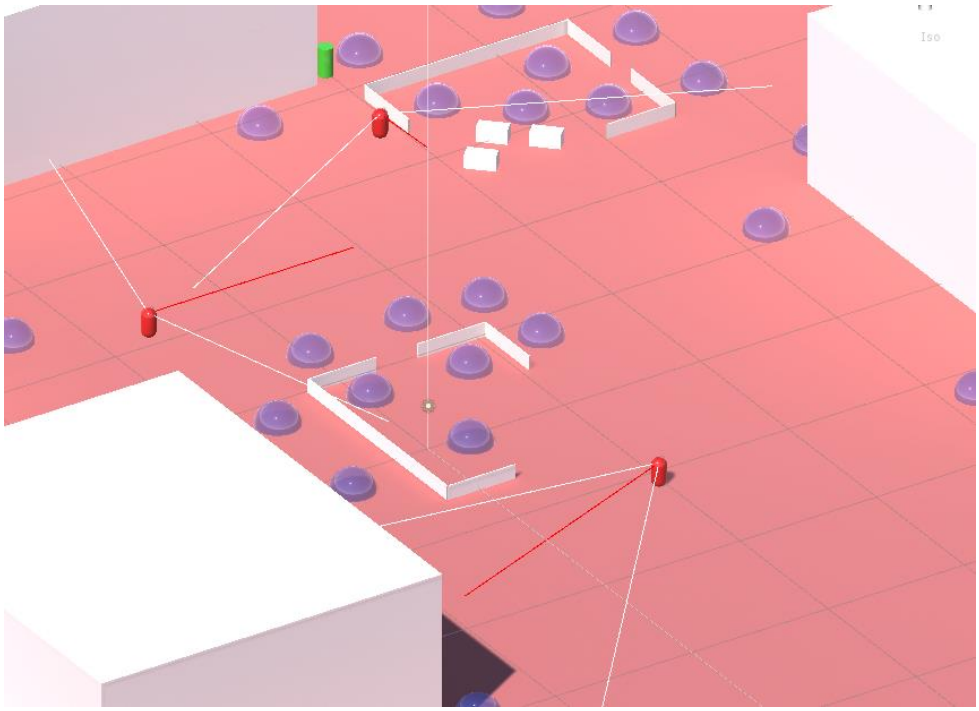
Dentro de este script vamos a tener una serie de parámetros:

- **attackDistance**, distancia a la que el enemigo puede atacar.
- **attackDamage**, daño que le produce al player.
- **chance**, variable que va a hacer que los enemigos tengan una probabilidad de fallo cuando disparan.
- **candency**, cadencia de disparo.
- **fireSound**, sonido de disparo.
- Y por último referencias a los scripts de CowBoyDecider y Health.

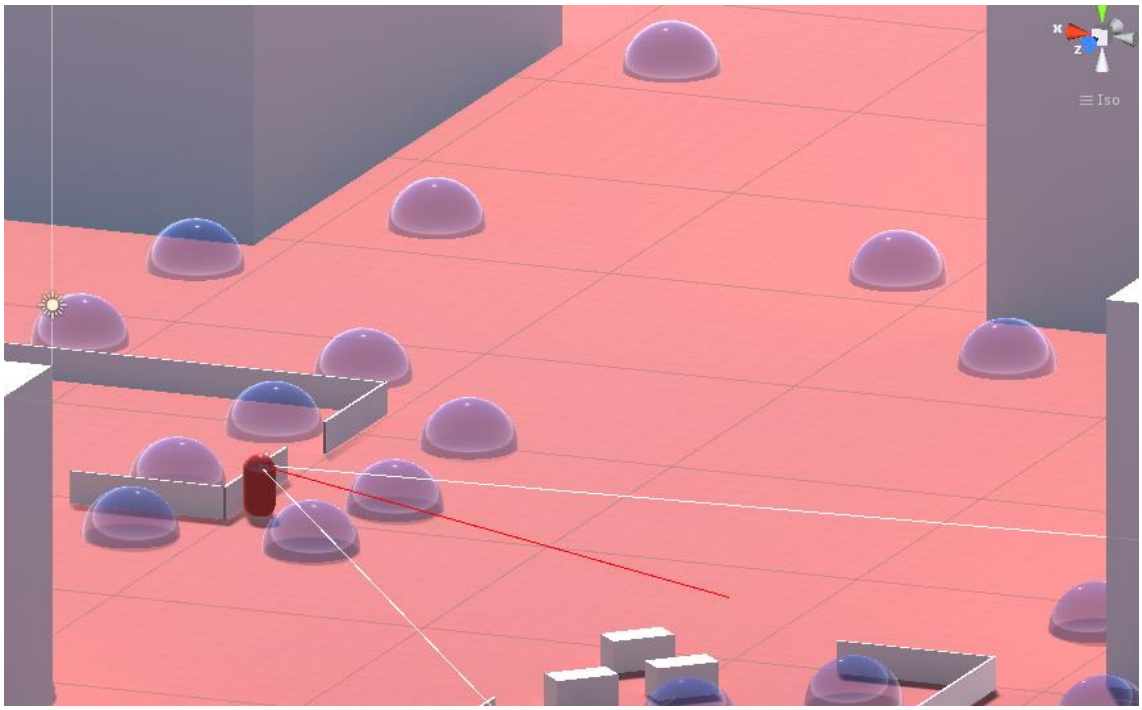
Teniendo la referencia de CowBoyDecider vamos comprobar cuando el enemigo está en estado de ataque y con la referencia a Health llamaremos a la función TakeDamage() pasándole por referencia el daño y la probabilidad de acierto. Al tenerlo de esta manera permite que cada enemigo tenga una cadencia de disparo y daño diferentes.

Por tanto este script nos permite comprobar que si el estado de los enemigos es attacking, estos atacarán restando vida al player mediante una corrutina para que no sea constantemente. Para que cada enemigo tenga una cadencia de disparo diferente llamaremos a la corrutina utilizando como valor de retorno la variable cadency creada en el propio script.

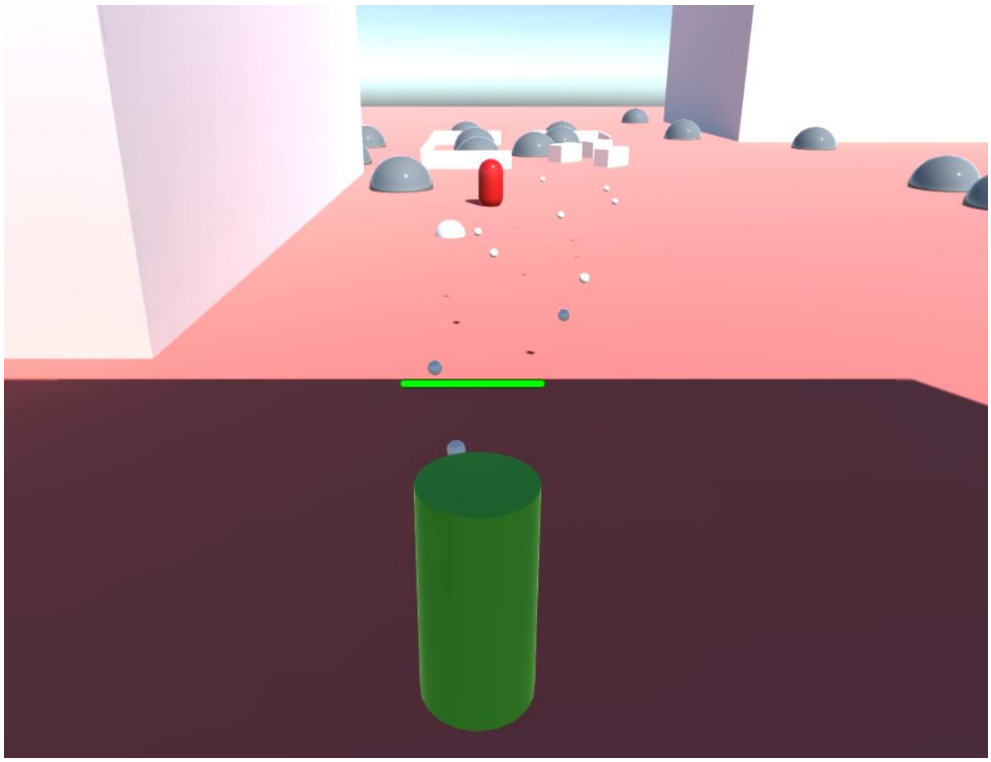
En este punto está implementado un controlador que gestiona el estado de los enemigos en todo momento. Estos comienzan patrullando, ya que al comenzar se le asigna una cobertura a la que ir, esta es random por lo que cada enemigo se mueve a una diferente. Si el player entra en su ángulo de visión estos pasarán a estar en modo wander y si consiguen llegar a estar en rango de tiro pasarán a modo attacking, en este estado cada enemigo tendrá un tipo de disparo con una probabilidad random de hacer daño al player y un daño específico cada uno. Un enemigo tendrá mayor rango de tiro y hará más daño pero al mismo tiempo tendrá una probabilidad mayor de fallar y una cadencia de disparo menor, otro tendrá un rango de tiro muy corto pero con una probabilidad muy alta de acertar y un daño mayor pero al mismo tiempo su candencia de disparo será muy reducida y el tercero y último estará en la mitad. De esta manera les damos un comportamiento diferente a cada enemigo.

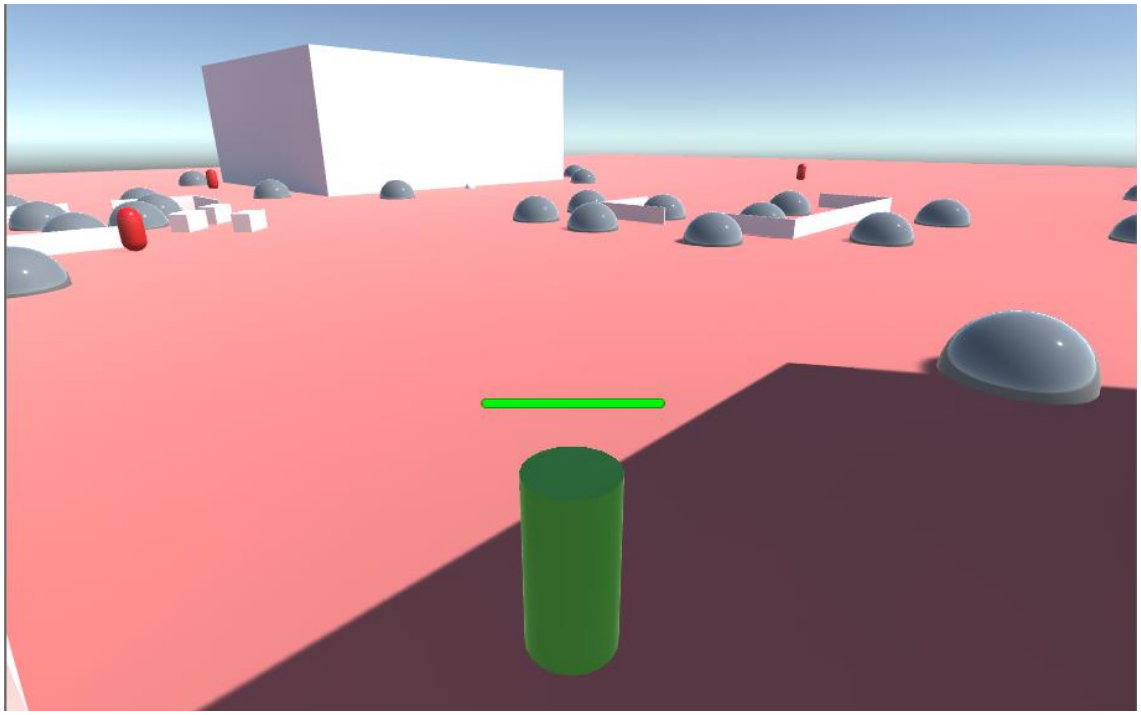


En caso de que el jugador deje de estar en su ángulo de visión los enemigos pasarán otra vez al estado de patrol para buscarle.



Antes de pasar a implementar el algoritmo, añadí una mecánica de disparo al player, al hacer click con el ratón disparará un proyectil en su Vector.forward, esto es una mecánica quise meter para continuar trabajando en el proyecto más adelante.





Implementación del algoritmo A*.

Para implementar el algoritmo partí de la primera parte de la entrega del laberinto, donde implementé gracias a tu ayuda el algoritmo de A*. Antes de comenzar la explicación creo conveniente puntuar que tenía que haber implementado primero el algoritmo antes que todo lo anterior explicado ya que una vez que conseguí hacer funcionar el algoritmo tuve bastantes problemas para combinarlo con el planteamiento anterior, el cual no estaba enfocado a introducir un algoritmo de búsqueda como puede ser A*. Por tanto fallo mío por un mal planteamiento desde un principio de la entrega. Aun que en mi defensa he de decir, que al no estar muy seguro de cómo iba a implementar el algoritmo opté por proporcionar una solución y trabajar en la entrega a no entregar nada.

Por tanto, paso a explicar la implementación del algoritmo.

Lo primero que hice cree fue una clase de tipo Nodo la cual iba a tener una serie de variables y métodos inicializados posteriormente en su constructor (Anexo 9). Una vez creada la clase de tipo Nodo, cree el propio script llamado Pathfinding.cs (Anexo 10.) el cual se le asignará a los enemigos, aquí he creado varios parámetros entre los cuales destaco un Vector3 que será el destino y dos listas donde vamos a guardar los nodos fronterizos y otra donde guardamos los nodos que serán el camino a seguir por el enemigo, es decir, los nodos de tipo padre.

Para realizar la búsqueda de nodos he creado un agente que será el que recorra todos los posibles nodos y devuelva la ruta que ha de seguir el enemigo, en vez de hacerla desde el propio enemigo. Una vez hecho esto he pasado a crear los métodos que conformaran el funcionamiento del algoritmo. El primer método será de tipo List<Nodo> donde calcularemos la ruta, este recibirá dos Vector3 como parámetros initialPosition y goal. A continuación inicializamos la posición del agente a initialPosition y a la variable destination (creada al inicio) le asignamos goal. Pasamos a la creación de un nuevo nodo, este recibirá como parámetros initialPsoition, 0 (coste), la distancia al goal y null (como padre).

Una vez que tenemos esto creamos otro método llamado GetFrontiers(), este recibirá un nodo del cual obtendremos 8 posibles posiciones a las cuales el enemigo se podrá mover. En este método lanzamos 8 raycast en las direcciones norte, noreste, este, sureste, sur, suroeste, oeste y noroeste. Para limpiar un poco el código refactoricé este método. Cada vez que creo el raycast para lanzarlo en la dirección específica llamo a otra función llamada TrowRayCast() que recibe un Vector3 con la dirección en la cual ha de lanzarlo y un nodo. En este método por medio de Raycast comprobamos con aquello que ha chocado el rayo lanzado tenga el tag “Ground”, en caso de que sea cierto creará un nodo en esa posición, al mismo tiempo comprobaremos si el tag es igual a player para cambiarle el comportamiento al enemigo. Una vez hecho esto, dentro de esta condición creo otro nodo pero en este caso cuando le paso la posición en la que el rayo ha chocado con el suelo, aumento el coste y le paso como padre currentNodo que es la variable de tipo nodo que le he pasado como referencia a este método. A continuación hago una comprobación para añadirlo o no la lista frontiers, solo lo añado si el padre es null o el nuevo nodo es distino al nuevo padre. Para hacer esta última comprobación he creado otro método llamado IsValidNode() de tipo bool, este recibe un parámetro de tipo nodo como referencia, este será comprobado para ver si está dentro de la lista o es igual a otro nodo ya creado, en caso de que esto sea verdad, devuelvo false, en caso contrario devuelvo true, lo que hace verdadera la sentencia dentro del método ThrowRayCast() y añadirá el nodo a la lista de frontiers. Antes de pasar esta lista al método calcualRuta() esta será ordenada en función de su prioridad (Hstar) por medio de esta sentecia:

```
frontiers = frontiers.OrderBy(nodo => nodo.priority).ToList();
```

Un vez que tenemos los nodos en la lista volvemos al método CalcularRuta(), donde vamos a hacer una serie de comprobaciones a la lista frontiers. Lo primero hacemos un bucle while donde solo nos aseguramos entrar si el número de elementos dentro de la lista es mayor que 0. En caso verdadero creamos un nuevo nodo al cual le vamos a asignar el nodo que está en la primera posición de la lista frontiers, como la lista está previamente ordenada nos aseguramos que sea el nodo con menor coste. A continuación asignamos la posición de este nodo al agente y creamos el último método con cual vamos a recorrer los padres RecorrerPadres(). Este método recibe un nodo como parámetro y devolverá una lista de nodos que recorrerá el enemy. Dentro del mismo creamos una nueva lista de nodos a la cual añadiremos los nodos padre. Antes de devolver esta nueva lista haremos un Reverse() de la misma para que al recorrerla empiece desde la posición correcta. Una vez hecho la retornamos y esta será asignada a la lista path creada al inicio del script Pathfinder.cs (Anexo 10). En la siguiente línea comprobamos si hemos llegado a goal en caso de que no sea cierto volvemos a hacer un llamamiento al método GetFrontiers() pasándole el nodo que estaba en la primera posición de la lista en vez de un nuevo nodo.

Anexo 12.

Y este sería el bucle que realiza el script, el agente recibir un destino, crea el camino hasta él y devuelve la mejora ruta, la cual es tomada por el enemigo para llegar a su destino.

Para una mejor visualización del algoritmo convertí el método en una corrutina, con la cual tenían un mayor control de lo que sucede en cada frame. Esto es meramente para visualizar mejor el funcionamiento del algoritmo.

Por último queda la implementación con el funcionamiento anterior. Para llevar esto a cabo he creado una referencia del script Pathfinder.cs (Anexo 10.) en CowBoyLocomotion.cs (Anexo 5.), en este script he creado dos varibles booleanas con las cuales voy a comprobar dos cosas: si nos estamos moviendo y si estamos calculando la ruta (moving, calculandoRuta). Por tanto si el enemy no se mueve quiere decir que estamos calculando una ruta, para calcular una ruta llamamos al método creado en Pathfinder -> CalcularRuta(), una vez que se esta ruta ha sido calculada llamamos al método moveTo() a través de la ruta calculada, ponemos a false calcularruta y moving a true, cuando hemos llegado a destino intercambiamos el valor de las variables booleanas. Esto se haría en bucle.

Anexo 13.

Anexo 14.

Anexo 15.

Discusión sobre los resultados obtenidos.

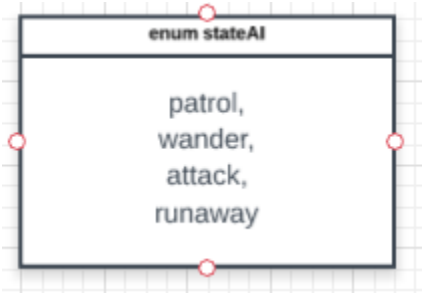
En general las sensaciones son buenas, conseguí corregir e implementar una gran cantidad de elementos a la entrega por lo que estoy bastante contento por ello. Respecto a mejoras con la entrega pasado puedo decir que las ha habido en todos los aspectos, conseguí corregir los errores que me daban al cambiar el estado de los enemigos y mejorarlo considerablemente, he añadido bastante nuevas funcionalidades para que sea algo jugable y entretenido y eso en nuestra carrera siempre hay que tenerlo en mente. Por lo que respecta al algoritmo, he de decir que fue un proceso bastante duro de implementar. Tuve problemas por todos lados, la primera fue que no sabía cómo iba a implementarlo, tuve en mente algunas ideas, por ejemplo, el dividir todo el escenario en cajas del mismo tamaño y hacer que fuera una cuadrícula. Pero la idea no me acababa de convencer aunque no dudo que fuera un planteamiento más sencillo que el que he llevado a cabo al final, no lo sé. El problema principal es que en el entregable del laberinto era en 2D y este era en 3D por lo que igual no iba a poder ser. Por lo que al final me decidí a plantearlo con un agente que iba lanzando raycast en 8 direcciones. El primer problema que tuve con esto fue el como método de lanzar los diferentes rayos, desde dónde, cómo iba a ir actualizando las posiciones, comprobar que posiciones iban a ser válidas, cuáles no. Más problemas que surgieron fueron al añadir los nodos a la lista frontiers, añadía demasiados, nodos repetidos por lo que el algoritmo no terminaba de convencerme. Poco a poco tras muchas pruebas y diferentes comprobaciones conseguí añadir los nodos correctos. Después para una mejor visualización implementé el método OndrawGizmos que me permitía ver los nodos de cada lista. Con esto me di cuenta que no reseteaba las listas una vez calculada la ruta por lo que se llenaban de miles de nodos super rápido, esto bajaba mucho la “perfomance”. Este último se solucionó rápido con una línea simple de código.

Por último decir que el algoritmo funcionaba bien por sí solo, es decir, antes de atacharlo al enemy y hacer que este se moviera. Pero al combinar ambos funcionamientos ha hecho que aparezcan más errores, como por ejemplo, que no hace más búsquedas después de realizar 3, o que cuando ve al player sí que lo detecta pero no vuelve a recalcular rutas si este sale de ángulo de visión.

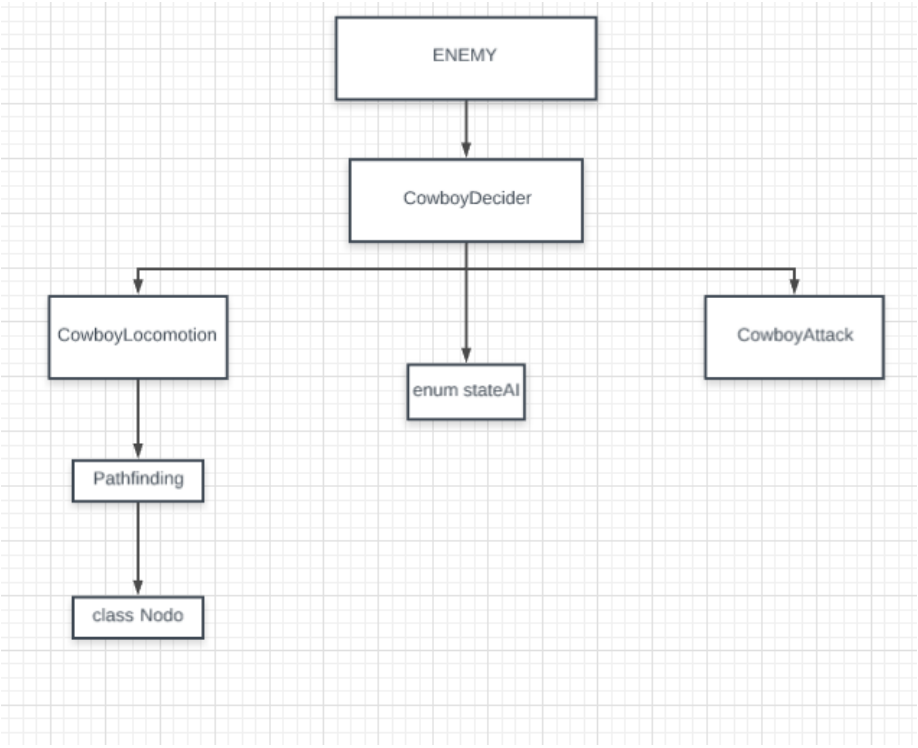
Como conclusión quiero decir que este proyecto me ha hecho mejorar mucho como programador, en un principio veía imposible el hacer funcionar un proyecto así. He aprendido mucho a lo largo del proceso y me has dado la base para yo poder continuar con este proyecto y mejorarlo considerablemente con un poco más de tiempo. Soy consciente de que esta entrega está muy lejos de ser, desde mi punto de vista, una buena entrega. Pero me quedo con la parte positiva que ha sido todo lo que he aprendido y el descubrir que echándole horas e intentarlo se pueden conseguir muchas cosas.

Anexos.

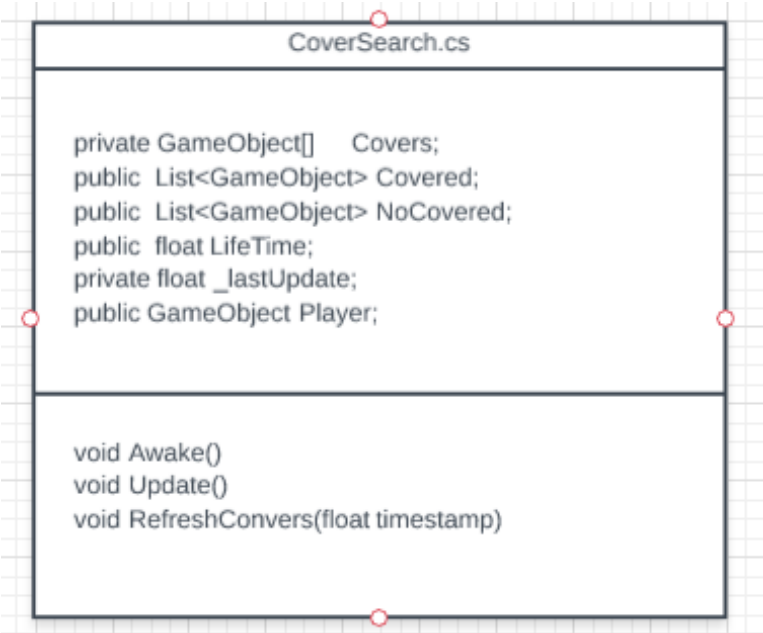
Anexo 1. Back. Back.



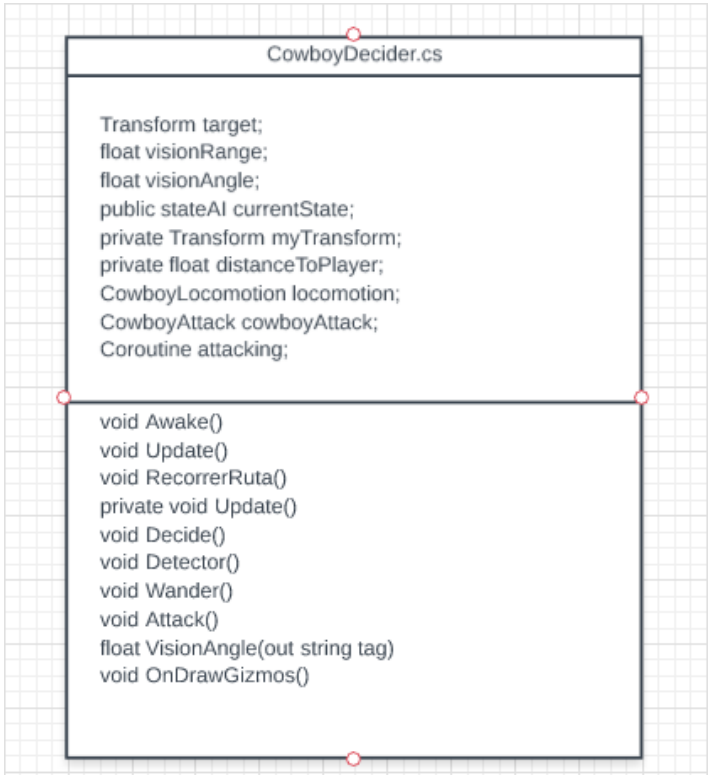
Anexo 2.



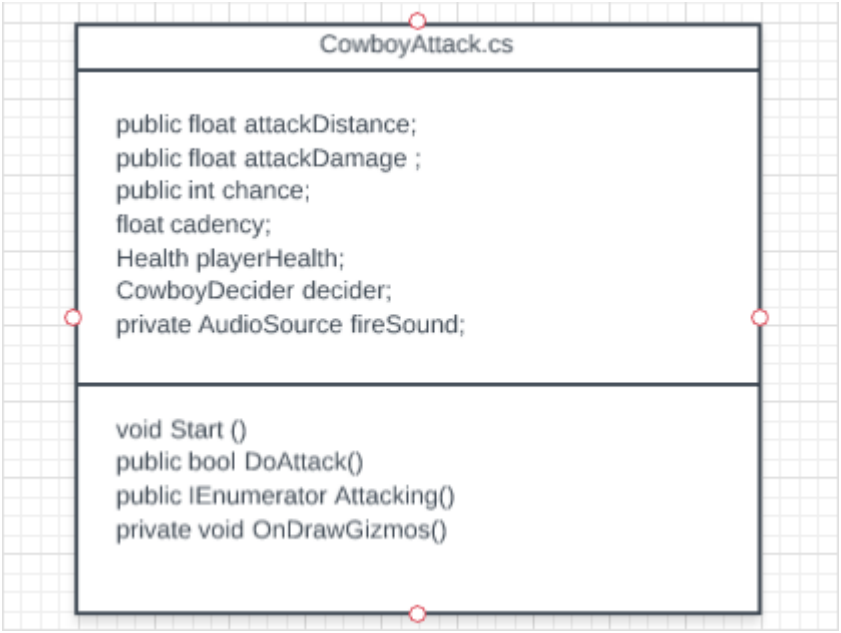
Anexo 3. Back.



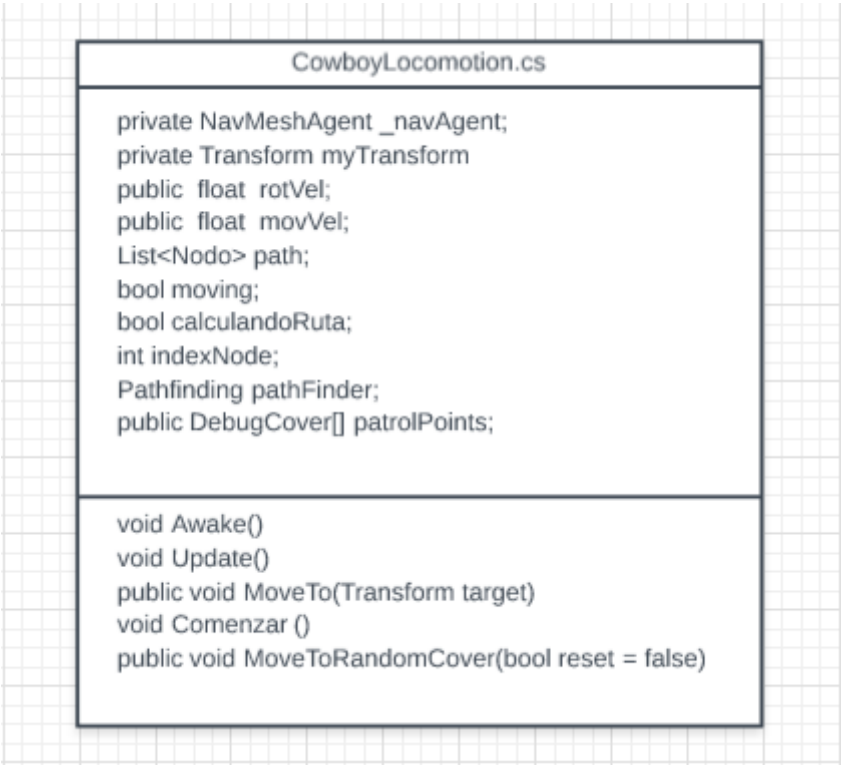
Anexo 4.



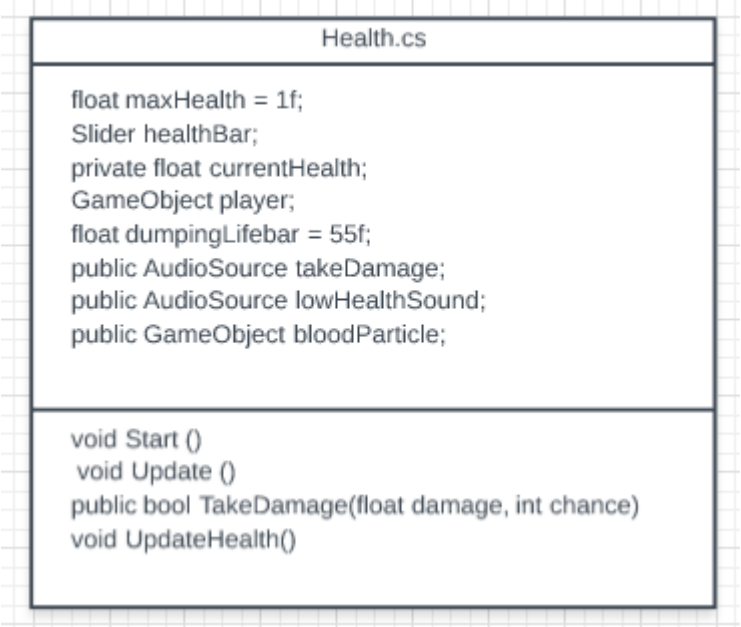
Anexo 5. Back. Back.



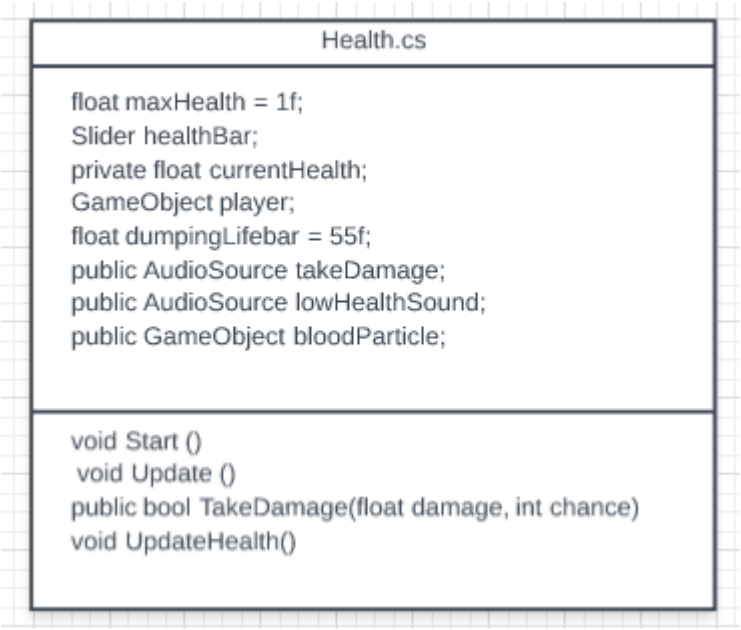
Anexo 6. Back.



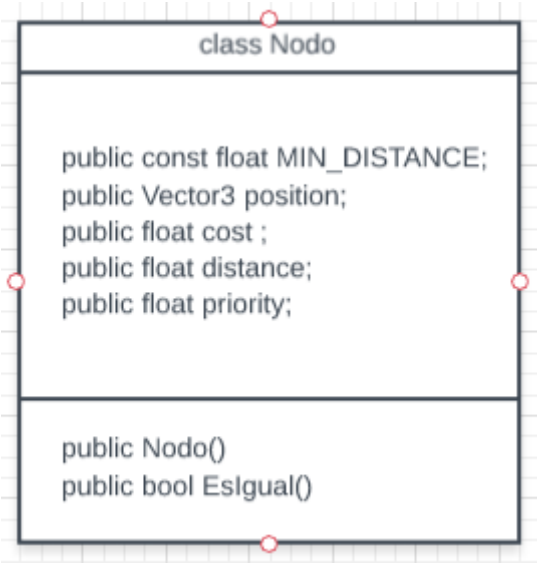
Anexo 7. Back..



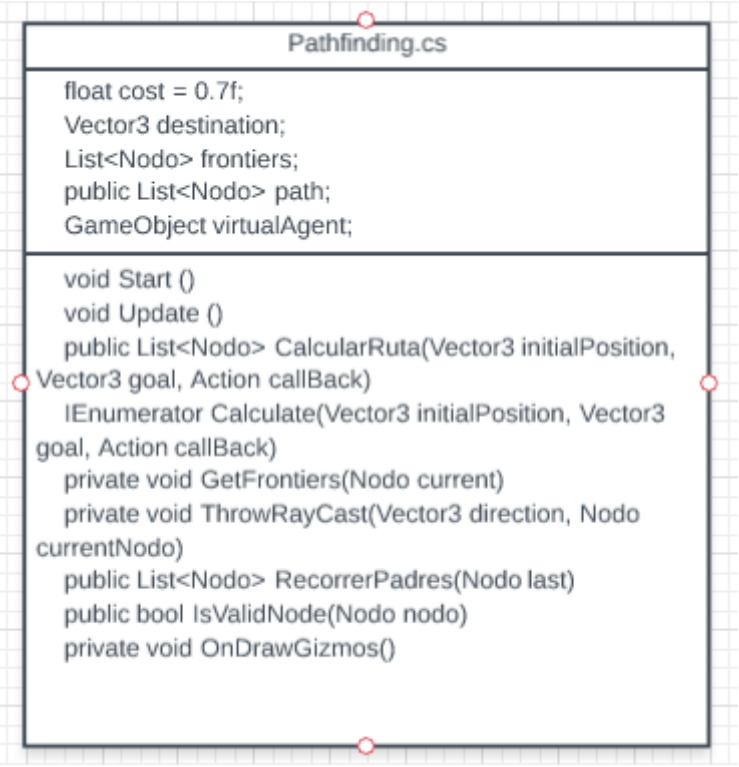
Anexo 8.



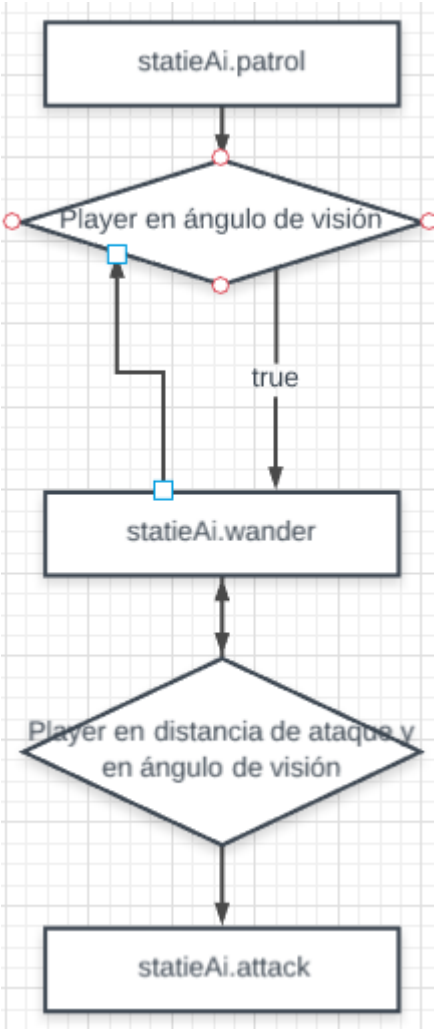
Anexo 9. Back.



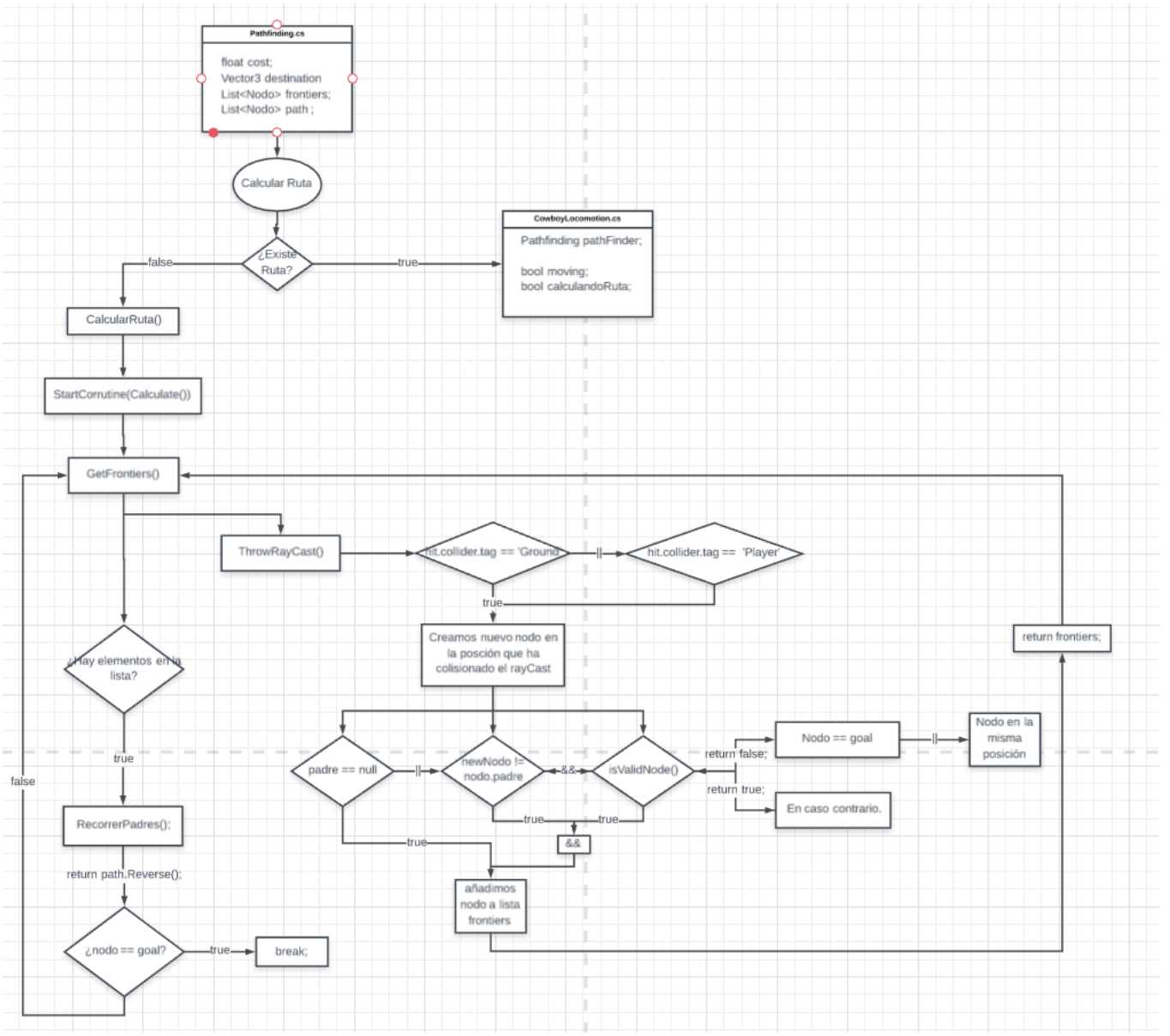
Anexo 10. Back. Back. Back.



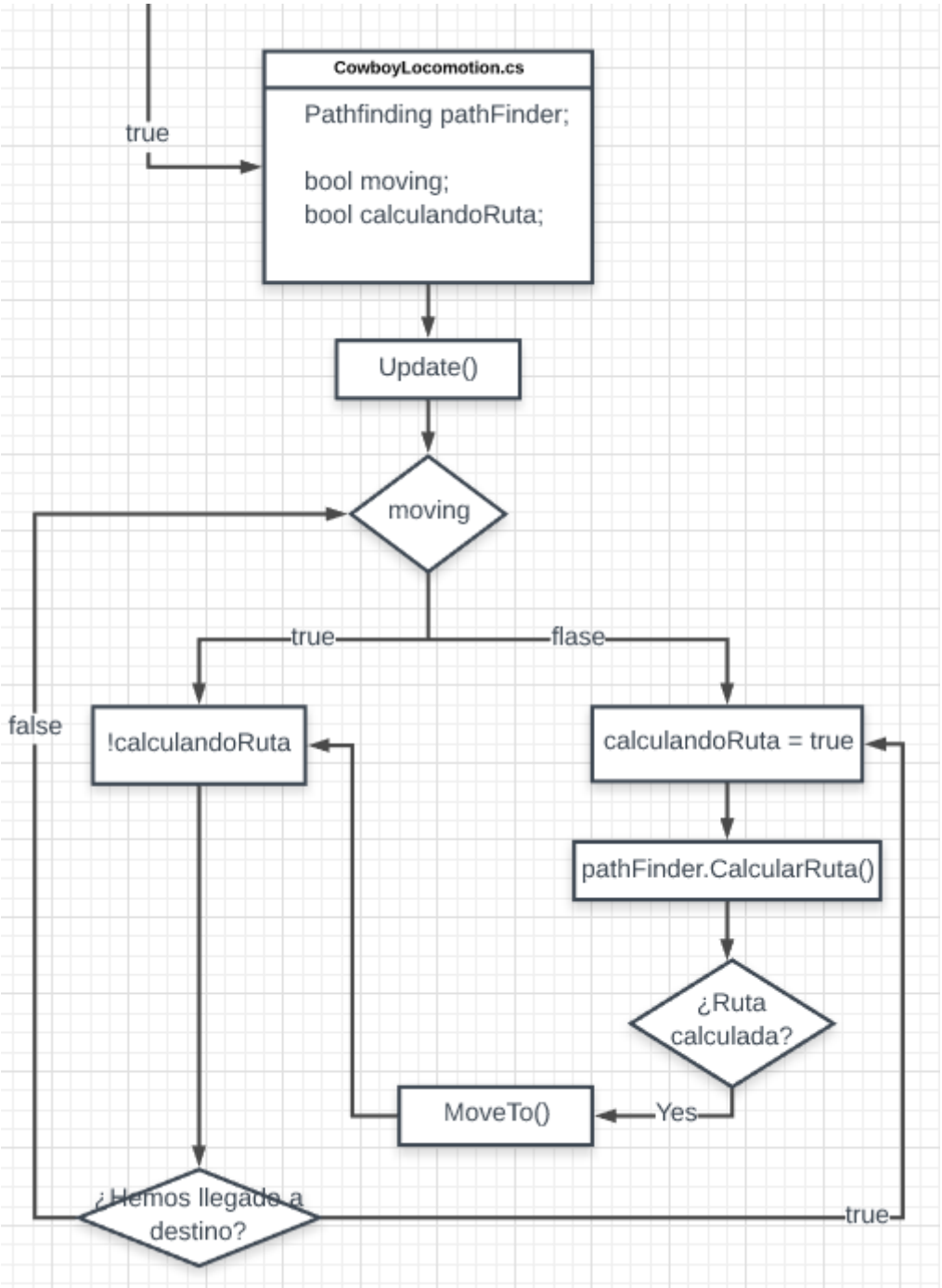
Anexo 11. Back.



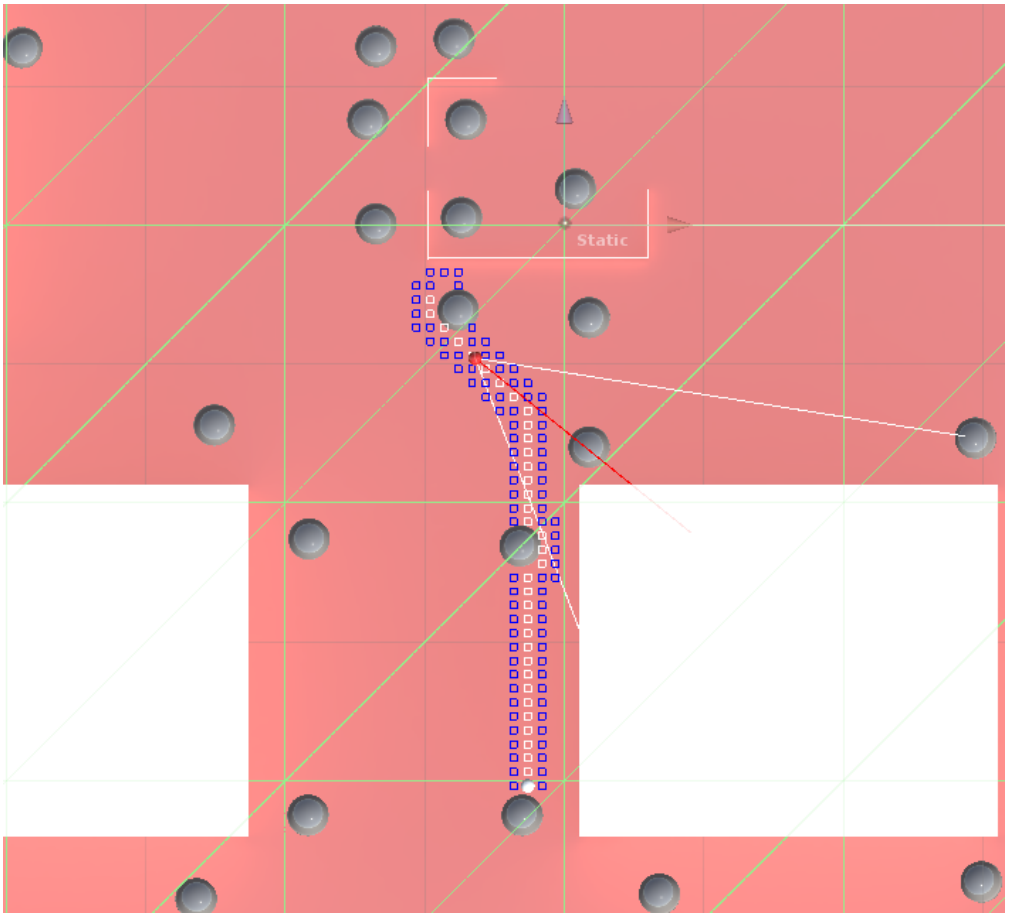
Anexo 12. Back.



Anexo 13. Back.



Anexo 14. Back.



Anexo 15. Back.