Guion Ejercicio Behavior Trees en Unreal

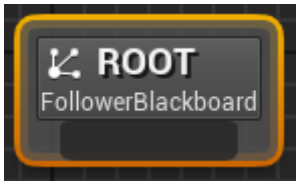
Parte 1: Repaso BehaviorTrees en Unreal

Hasta ahora hemos visto un poco como con los blueprint se podían montar comportamientos más o menos sencillos, pero a la hora de crear una IA más compleja lo mejor es usar behavior trees. Los behavior trees nos ofrecen una manera mucho más estructurada para generar comportamientos y modificarlos fácilmente.

Primero vamos a recordar los nodos que se pueden usar en los Behavior Trees pues usaremos BTs a partir de ahora:

Root

El nodo inicial del árbol, la raíz como su propio nombre indica, todos los BT comienzan con un nodo Root. El nodo Root no tiene propiedades pero si los seleccionamos podremos ver las propiedades del BT.



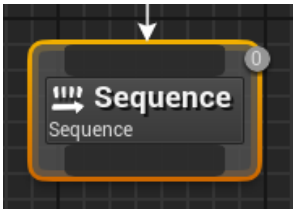
Composites

Son el punto de partida de los BTs; debajo del Root siempre tendremos que poner un nodo Composite. Tiene N hijos y definen como es el flujo del árbol y cómo se comporta en base a los resultados obtenidos.

Sequence

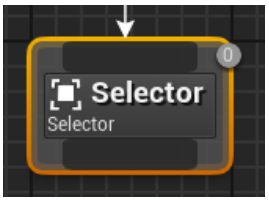
Un nodo composite que lo que hace es ir ejecutando uno a uno sus hijos de izquierda a derecha mientras estos terminen exitosamente. Es decir sirve para secuenciar cosas, por ejemplo si estoy en mi casa y quiero ir a comprar el pan tendré la secuencia: Coger dinero –

Salir de casa – Ir a la panadería – Comprar el pan – Volver a casa. Este nodo terminara con éxito si todos sus hijos terminan con éxito, en cuanto uno falle terminara con fallo.



Selector

El nodo selector es otro nodo composite pero se diferencia de la secuencia en que va ejecutando sus hijos de izquierda a derecha mientras fallen. Es decir intenta buscar uno que se ejecute con éxito y en tal caso terminara su ejecución exitosamente. Si no encuentra ninguno terminara con fallo.



Simple Parallel

El ultimo nodo composite y nos permite ejecutar una Task y un árbol a la vez. Se usa para crear árboles que necesitan de otra tarea ejecutándose a la vez. Este nodo se ejecutará hasta que la Task termine.

Decorators

Los decoradores son condiciones que se pueden añadir a los nodos para controlar su ejecución, por ejemplo, comprobar algo en la blackboard, una probabilidad o cualquier tipo de condición que se nos ocurra. Por ejemplo, si el target está a una determinada distancia.

Services

Los servicios se ejecutan sobre los nodos composite mientras estos se estén ejecutando y modifican el estado de la IA. Por ejemplo, puedes tener un servicio en un decorador que busque al target y lo meta en una variable de la blackboard para que los nodos hijos lo usen.

Tasks

Hacen todo el trabajo, se ejecutan y al terminar devuelven éxito o fallo para que los composites a los que estén conectadas actúen en consecuencia. Un ejemplo muy típico de Task en el nodo Move To para mover al Character a donde necesitemos. En este caso el Task podría permanecer en ejecución varios ciclos convirtiéndose en un Task latente.

Una vez revisados los nodos que se pueden utilizar en un BT lo siguiente que necesitamos es saber que es una blackboard. Una **blackboard** no es más que un sitio donde apuntar valores y que se comparte por todos los nodos del behavior tree y hacia el exterior, de forma que cualquiera podría modificarla si se lo permitimos.

Entonces lo que vamos a tener es un **Character** que representa la entidad en el mundo, un **AIController** que nos ofrece funcionalidad para mover ese Character y para ejecutar un Behavior Tree. Un **Behavior Tree** y una **Blackboard** para la comunicación de los nodos.

Parte 2: Chase al player sobre un BehaviorTree usando BluePrint

El objetivo es tener un Character controlado por la IA que esté parado en su posición de spawn y cuando vea al player a una distancia le siga hasta perderle de vista. Entonces volverá al punto de spawn.

Esta vez vamos a usar el TopDown Template, por cambiar un poco, aunque se podría usar el ThirdPerson si quisiéramos.

Antes de empezar vamos a crear un **NavMeshBoundsVolume** como hicimos en la parte 4 para crear la NavMesh y que los Characters puedan usarla para su navegación.

Ya que tenemos NavMesh ahora vamos a crear el resto de elementos necesarios:

1. Creamos nuestro **Character** AI_Character como hicimos el primer día.
2. Creamos un nuevo **AIController**, tal como hemos hecho anteriormente, al que llamaremos MyAIController.
3. Creamos un **Behavior Tree**, para ello hacemos clic derecho en la carpeta de los blueprints del **Content Browser** y lo creamos seleccionándolo en el menú **Create Advanced Assets/Artificial Intelligence/Behavior Tree**. Lo llamamos MyBehaviorTree.
4. Por último creamos la Blackboard que el igual que el Behavior Tree está en **Create Advanced Assets/Artificial Intelligence/Blackboard**. La llamamos MyBlackboard.

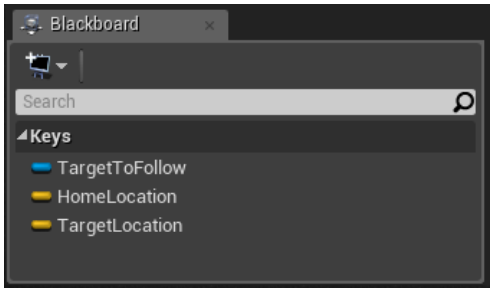
Una vez creados todos los assets que vamos a necesitar (de momento) vamos a arrastrar un MyAICharacter al mapa para tener una instancia y a configurar el resto.

Blackboard

La Blackboard la vamos a utilizar para apuntar la información necesaria que será referenciada por el BT. En este caso vamos a necesitar las siguientes variables:

a. *TargetToFollow*: El target al que debemos perseguir.
b. *HomeLocation*: La posición donde volver al perder al target.
c. *TargetLocation*: La última posición conocida el target.

Hacemos doble clic en la Blackboard para abrir su editor y configuramos las variables:



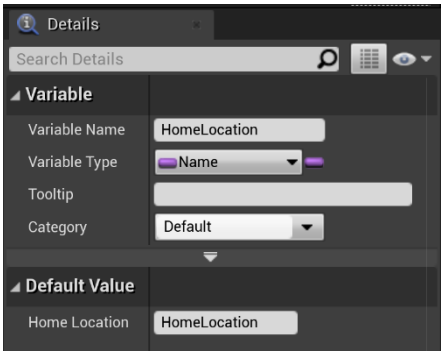
1. Usando el botón New Key añadimos una nueva entrada tipo Object que se llame TargetToFollow.
2. Añadimos otra tipo Vector llamada TargetLocation.
3. Añadimos otra tipo Vector llamada HomeLocation.

AIController

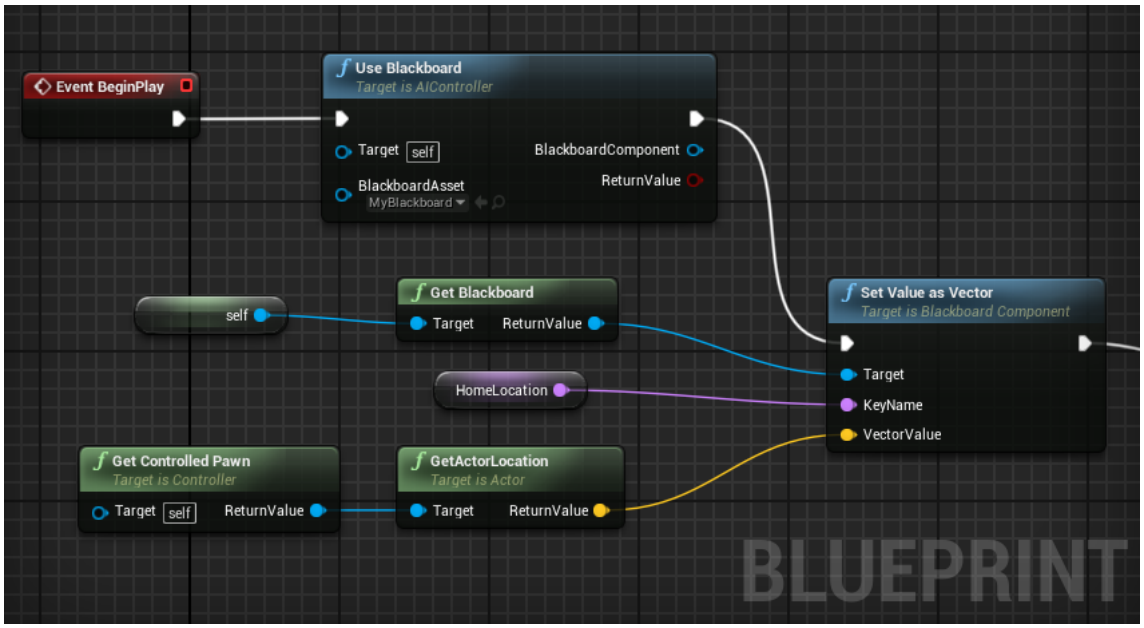
Para editar el AIController "MyAIController" hacemos doble clic sobre él y abrimos el **EventGraph**. Vamos a inicializar el AIController y a lanzar la ejecución de BT.

1. Creamos un evento **Begin Play** (solo se lanza cuando el AIController entra en juego) y lo vamos a utilizar para la inicialización.
2. Añadimos un nodo **Use Blackboard** y lo configuramos para que use una blackboard del tipo **MyBlackboard**.

3. Creamos una variable de tipo **Name**, llamada **HomeLocation**, que tenga el valor por defecto **HomeLocation** (si nos os dejara asignarle un valor a la variable había que compilar el blueprint). Nota: en versiones anteriores de Unreal, esta variable podía ser de tipo **String**, ya que el nodo **Set Value as Vector** que usaremos a continuación recibía un **String** como parámetro de entrada, pero hoy en día recibe una variable de tipo **Name**. Este es el tipo de cambios de los que os he hablado para que os hagáis una idea de los problemas que pueden surgir al actualizar nuestro proyecto a nuevas versiones del motor.



4. Y después tenemos que añadir a la blackboard la posición de spawn (HomeLocation), para ello usamos el nodo **Set Value as Vector**, como entrada recibe la **Blackboard** que se la pedimos a **self**, **HomeLocation** que creamos anteriormente como **KeyName** y con **Get Controlled Pawn** y **Get Actor Location** sacamos la posición y se la metemos a **VectorValue**.



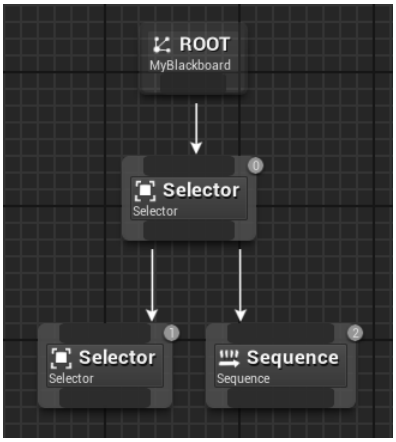
5. Por último tenemos que lanzar la ejecución del BT. Para ello usamos el nodo **Run Behavior Tree** y le configuramos el **MyBehaviorTree**.

Behavior Tree

Vamos a crear los nodos del BT para modelar el comportamiento deseado: hacemos doble clic sobre MyBehaviorTree y se abrirá el editor. Nada más abrirlo tendrá un nodo Root; es el nodo inicial de todos los árboles y estará creado por defecto. Tiene la particularidad de que no puede ser hijo de ningún otro y solo puede tener un hijo.

Podemos crear nuevos nodos fácilmente, con botón derecho o arrastrando desde debajo de un nodo. Nos saldrá un menú contextual con los nodos que podemos crear: los composites, los decorator, las tasks y los services.

De momento vamos a crear un **Selector** debajo del **Root** y debajo de este Selector un **Selector** y una **Sequence**.

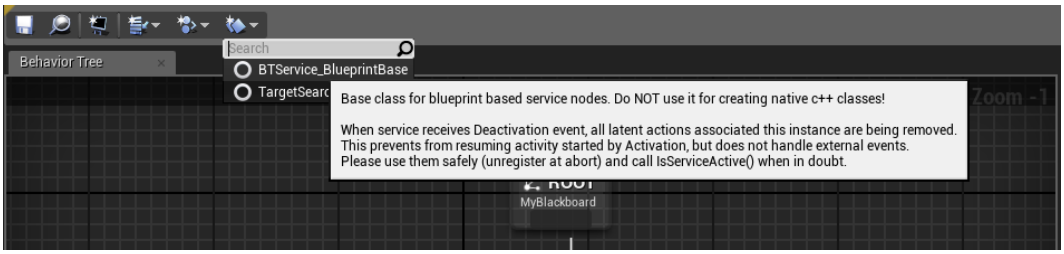


Ya tenemos la estructura básica que va a tener nuestro BT pero ahora hay que crear toda la lógica que necesitamos para que funcione como queremos.

Service

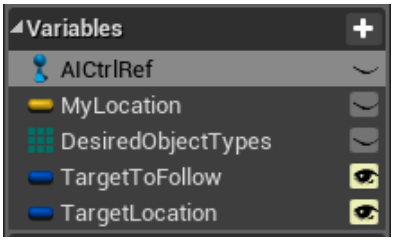
Lo primero que necesitamos es un **Service** que se encargue de comprobar si vemos al jugador; lo hará tirando un rayo al jugador como vimos en las prácticas del primer día. En el caso de que lo estemos viendo vamos a escribirlo en la blackboard, en la variable **TargetToFollow**, de forma que en el BT podamos comprobar fácilmente si vemos o no al jugador solo mirando el valor de esa variable.

¿Cómo creamos un **Service en BluePrint**? Pues muy fácil o usamos el menú de arriba del editor de BT y seleccionamos **BTService_BlueprintBase**, o lo creamos como hemos creado blueprints anteriormente, con el menú de nuevo blueprint y buscamos el tipo **BTService_BlueprintBase**.

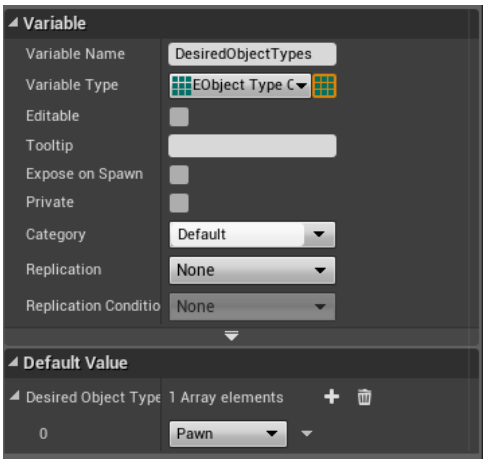


Lo llamamos **TargetSearch** y hacemos doble clic sobre el para editarlo.

1. Creamos las variables que vamos a usar en el servicio:

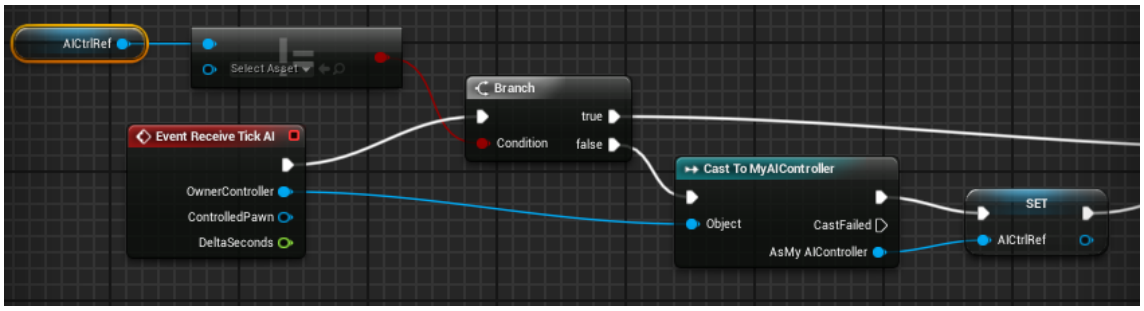


   a. *AICtrlRef*: Una referencia a un objeto del tipo MyAIController, para almacenar nuestro controlador.
   b. *MyLocation*: Variable Vector donde guardaremos nuestra posición para poder consultarla en el Blueprint.
   c. *DesiredObjectTypes*: Un **array** del tipo **EObject Type Query** con una entrada tipo Pawn como valor por defecto (para poder configurar el valor por defecto hay que compilar el blueprint). La usaremos como entrada del nodo **MultiSphereTraceForObjects** que usaremos para obtener todos los actores interesantes en un determinado radio.

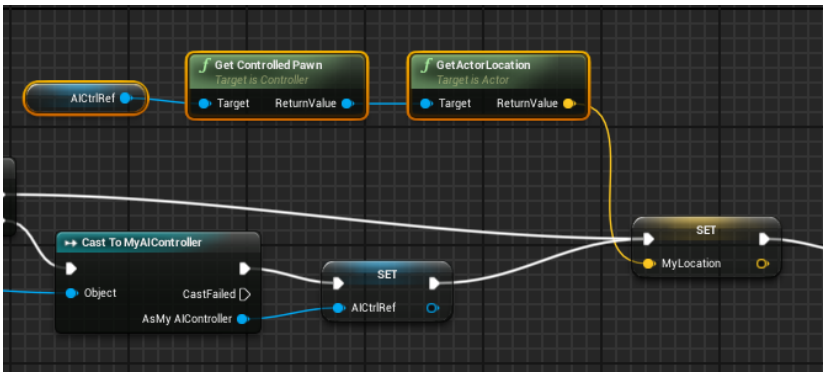


   d. *TargetToFollow* y *TargetLocation*: Dos variables de tipo **Blackboard Key** que usaremos como claves para acceder a la blackboard. Estas variables las vamos a poner como **Editables** para que se puedan configurar desde el BT.

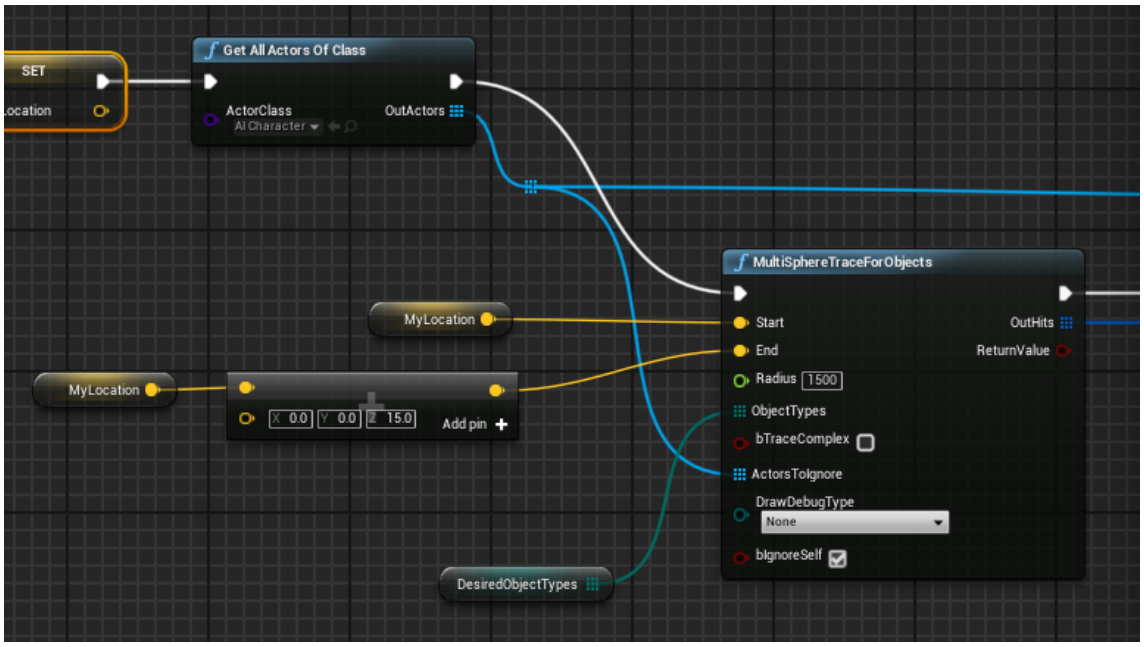
2. El punto de entrada del "sensor" será un evento del tipo **Event Receive Tick AI**, este evento se llamará cada tick del servicio (dependiendo de la configuración que tenga en el BT) y del podremos obtener el **OwnerController**, a diferencia del **Event Receive Tick** que no lo tiene.
3. Lo primero va a ser mirar si la variable **AICtrlRef** tiene un valor y si no es así hacemos un **Cast to MyAIController** al **OwnerController** del evento y se lo asignamos a la variable **AICtrlRef**. Para ello cogemos la variable **AICtrlRef** miramos si es distinta de NULL con el nodo **!=** y utilizamos un nodo **Branch** para llamar al **Cast to MyController** y hacer el **SET** de la variable o continuar la ejecución normalmente.



4. Para que todo quede un poco más limpio vamos a guardar la posición actual en la variable **MyLocation**, conectaremos el true del **Branch** y el **SET** a un nodo **SET** con la variable **MyLocation**. Como entrada **AICtrlRef** con un **Get Controller Pawn** y un **Get Actor Location**.

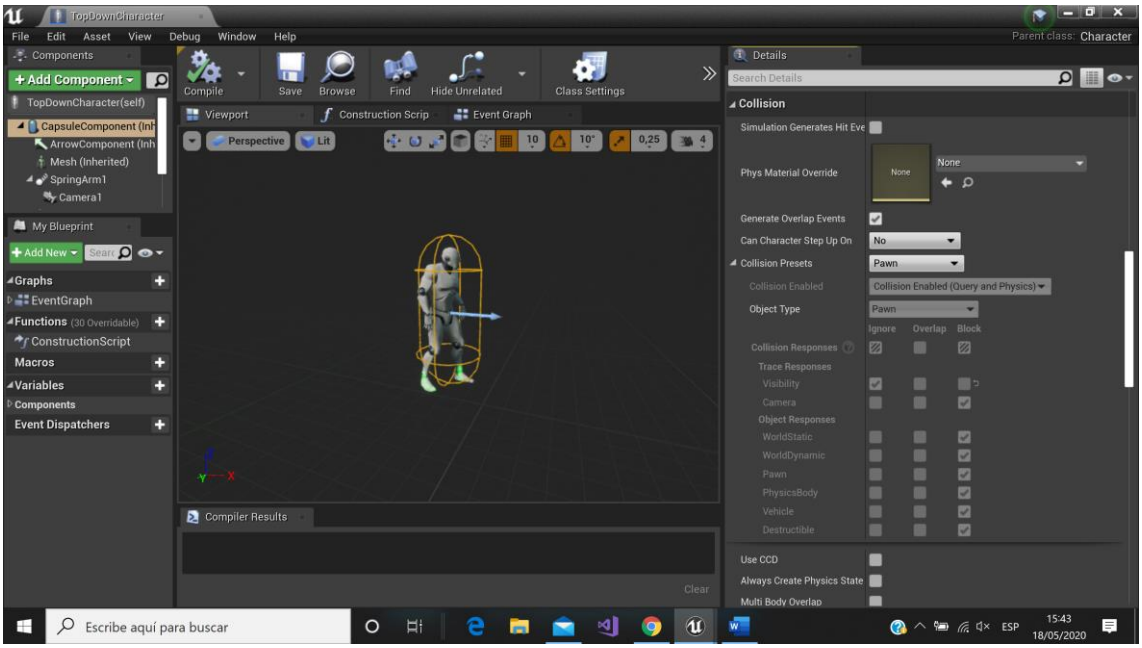


5. Ahora tenemos que utilizar la física para ver si detectamos al jugador, para ello vamos a utilizar en este caso un **MultiSphereTraceForObjects** que nos va a devolver todos los actores del tipo seleccionado que este a la distancia deseada. Pero antes debemos tener todos los datos que necesita este nodo como entradas:
    a. *Start*: Variable **MyLocation**
    b. *End*: Variable **MyLocation + Vector(0,0,15)**
    c. *Radius*: Vamos a configurar **1500**.
    d. *ObjectTypes*: La variable **DesiredObjectTypes** que configuramos anteriormente con una entrada tipo **Pawn**. Esto será el tipo de actores que busque el nodo.
    e. *ActorsToIgnore*: Queremos que ignore todos los Pawn que tengan asignado un AIController así que vamos a usar el nodo **Get All Actors Of Class** y le configuramos **AICharacter**.

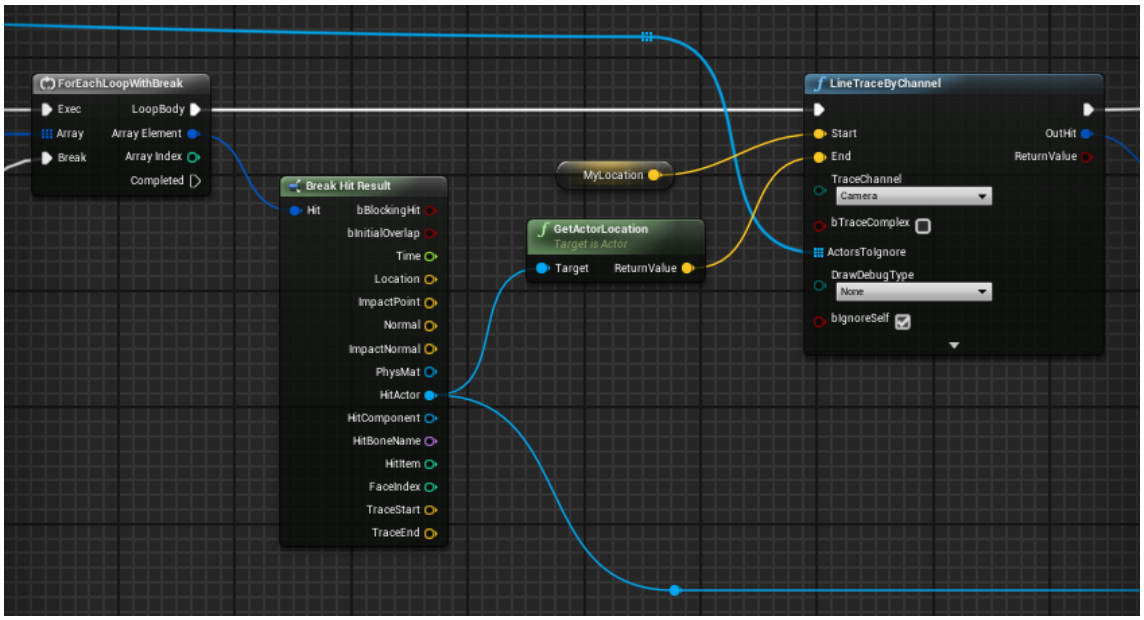


6. Como resultado tenemos en **OutHits** todos los hits encontrados; lo que vamos a hacer es iterar estos hits para ver si vemos al jugador o no tirando un rayo a su posición. Vamos a usar un nodo **For Each Loop With Break** para iterar cada uno de los hits y vamos a tirar un rayo desde **MyLocation** hasta la posición del actor encontrado por el hit para ver si le vemos realmente.
    a. Usamos un nodo **For Each Loop With Break** y como entrada le metemos el array de hits de **MultiSphereTraceForObjects**.
    b. En cada **LoopBody** hacemos un **LineTraceByChanel** como el que usamos anteriormente en la Parte 2, pero esta vez vamos a utilizar el **OutHit** para calcular más cosas.
    c. Como entradas de **LineTraceByChanel** tenemos que meter:
        i. **Start: MyLocation** será el origen del rayo
        ii. **End:** Final del rayo: La posición del actor que estamos iterando, para ello usamos un nodo **Break Hit Result** que nos saca toda la información del hit y conectamos **Get Actor Location** a la salida **HitActor**

        iii. **TraceChannel:** Camera… ¿por qué Camera? Miramos el ThirdPersonCharacter que se creó por defecto en la escena, es el player al que nuestra IA perseguirá. En su CapsuleComponent, en el apartado Collision vemos que tiene un CollisionPreset Pawn; A continuación, nos muestra los flags de colisión de dicho preset y vemos que Ignore a Visibility pero Block con Camera.

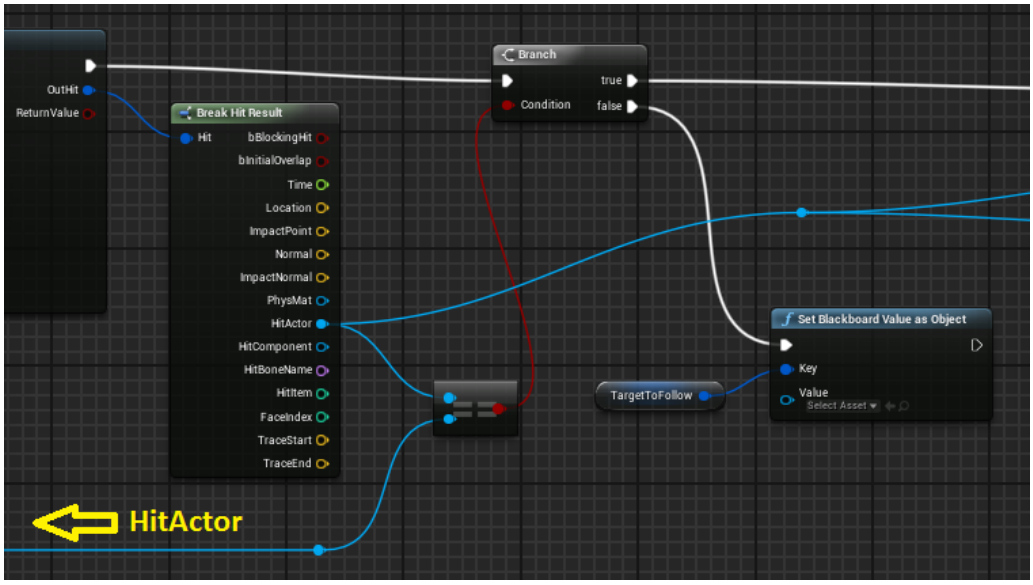


iv. **ActorsToIgnore**: el array de AIControllers que calculamos anteriormente y que usamos para los testeos de esfera.

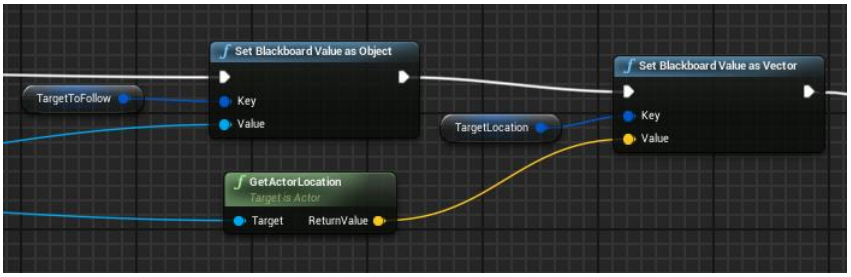


NOTA: Observar que el **For Each Loop With Break** tiene una entrada configurada en el Break: esto lo haremos cuando consigamos detectar al target y escribamos los valores en la blackboard para que no siga iterando por los hits.

Ahora tenemos que manejar el resultado del **Line Trace By Channel** para comprobar si realmente vemos al jugador o no. Lo primero va a ser usar un nodo **Break Hit Result** para sacar la información del hit y comprobar que el actor con el que colisionamos sea el mismo que estamos iterando. Comparamos los dos actores con un nodo **==** y llevamos el resultado a un **Branch**. Si son iguales continuamos, sino escribimos **NULL** en la variable **TargetToFollow** de la blackboard. Para hacer esto usamos un nodo **Set Blackboard Value as Object** y no le configuramos valor, de modo que se pondrá a NULL. Como entrada usamos la variable tipo **Blackboard Key** que creamos anteriormente **TargetToFollow**.



En caso de ser el mismo actor debemos escribir ese actor en la variable **TargetToFollow** de la blackboard con el nodo **Set Blackboard Value as Object** usando como key **TargetToFollow**. Además, vamos a escribir en la blackboard la variable **TargetLocation** con un nodo **Set Blackboard Value as Vector** para tener almacenada la última posición donde vimos al target para poder ir a dicha posición una vez le perdamos de vista.



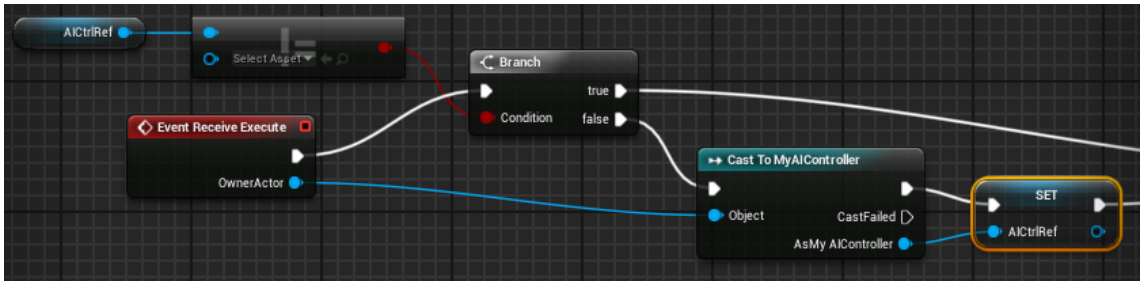
Para terminar, como comentábamos antes, conectamos la salida del **Set Blackboard Value as Vector** al **Break** de **For Each Loop With Break** para no seguir iterando una vez encontrado el target.

Con esto ya tenemos nuestro Servicie que busca al target y lo escribe en la blackboard (**TargetToFollow**) cuando lo encuentra o borra de la blackboard el valor cuando lo pierde. Además, escribe en la blackboard (**TargetLocation**) la última posición conocida del target para poder ir a ella al perderle.

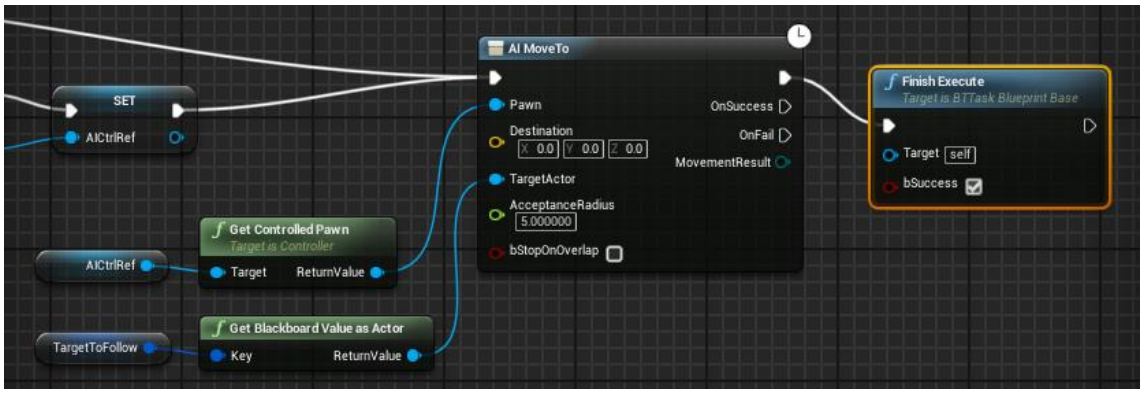
Task

También necesitamos crear un nuevo **Task,** ya que no hay ninguna por defecto que vaya hasta un Actor; para ello, al igual que con el servicio, vamos a: crear una Task nueva (**BTTask_BlueprintBase**) y la vamos a llamar **MoveToActor**. Hacemos doble clic para editarla.

1. Lo primero que vamos a hacer es configurar las variables de la Task:
    a. ***AICtrlRef***: Una referencia a un objeto del tipo **MyAIController**, para almacenar nuestro controlador.
    b. ***TargetToFollow***: Variable de tipo **Blackboard Key** que usaremos como clave para acceder a la blackboard. Esta variable las vamos a poner como **Editable** para que se pueda configurar desde el BT.
2. El punto de entrada de nuestra **Task** será el evento **Event Receive Execute** que se llamará cada tick.
3. Lo primero que hacemos es guardar el controlador en la variable **AICtrlRef** tal como hicimos en el servicio.



4. Después ejecutamos el movimiento con un **AI MoveTo**, cogemos nuestro Pawn de **AICtrlRef** con **Get Controller Pawn** como **Pawn** de entrada y el **Actor** almacenado en la variable **TargetToFollow** como destino, usando **Get Blackboard Value as Actor** con la variable **TargetToFollow** de entrada.
5. Para terminar, usamos **Finish Execute** cuando el **AI MoveTo** termine y lo marcamos como **bSuccess**.



Ya tenemos nuestra tarea que lo único que hace es moverse hasta el actor que esté en la variable TargetToFollow que le configuremos.

IMPORTANTE: Que la variable se llame **TargetToFollow** no hace que se referencie la variable **TargetToFollow** de la blackboard: este es el nombre que tiene en la task esta Blackboard Key;

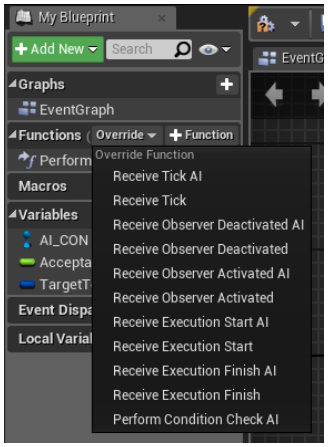
se suele usar el mismo nombre por claridad, pero a la variable que referencia se configura en el editor de BTs al crear la task.

Decorator

Ya hemos creado nuestro **Service** para detectar al target y una **Task** para ir a la posición de un actor; solo nos queda crear un **Decorator** para comprobar la distancia al target y empezar la persecución.

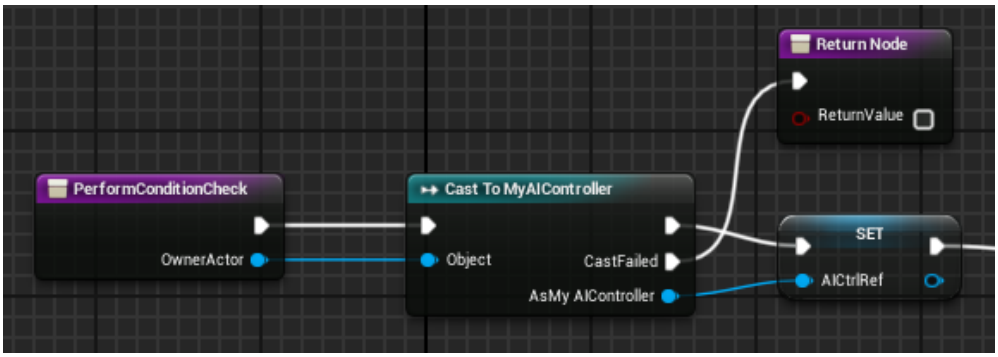
Al igual que con el **Service** y la **Task** creamos un nuevo **Decorator** (BTDecorator_BlueprintBase) y lo llamamos **CheckTargetDistance**. Una vez lo hemos creamos hacemos doble clic para editarlo.

1. Primero tenemos que configurar a qué tipo de función vamos a hacer override para que se llame al decorador y poder devolver un resultado. En la pestaña **My Blueprint**, en la sección **Functions,** seleccionamos **Override** y en el menú desplegable seleccionamos **Perform Condition Check**. Esto nos permite recibir un evento cuando se esté comprobando el evento de condición.

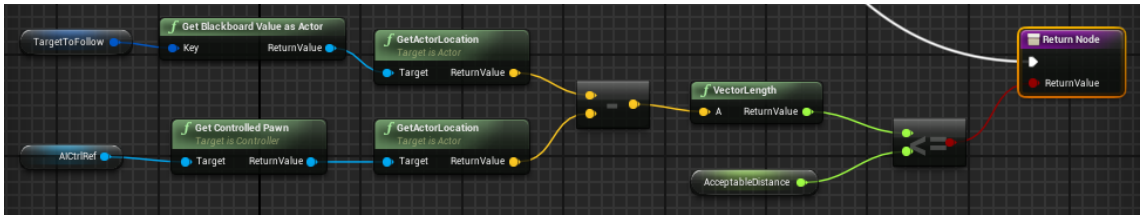


2. Configuramos las variables del evento:
    a. ***AICtrlRef***: Una referencia a un objeto del tipo **MyAIController**, para almacenar nuestro controlador.
    b. ***TargetToFollow***: Variable de tipo **Blackboard Key** que usaremos como clave para acceder a la blackboard. Esta variable la vamos a configurar como **Editable** para que se pueda modificar desde el BT.
    c. ***AcceptableDistance***: Variable tipo float para configurar la distancia a la que hacer el check con el target. Marcamos esta variable como **Editable** para poder configurarla en el BT.

3. Primero hacemos un **Cast to MyAIController** al **OwnerActor** del **Perform Condition Check;** en caso de fallar el cast hacemos un **Return Node** con **ReturnValue** a false para devolver que el target no está dentro del **AcceptableRadius**. En caso de poder hacer el cast guardamos la variable en **AICtrlRef** y hacemos la comprobación de radio.



4. Para comprobar si está dentro del radio simplemente tenemos que restar las posiciones del **Pawn** y del **Target** y ver si el tamaño es menos que el **AcceptableRadius**, para ello cogemos de la blackboard el **TargetToFollow** con **Get Blackboard Value as Actor** y le pedimos su posición con **Get Actor Location** y a **AICtrlRef** le pedimos su **Get Controlled Pawn** y de nuevo su **Get Actor Location**. Restamos los dos vectores con **–** y sacamos su tamaño con **VectorLength**.
5. Ya solo nos quedaría comprobar si esta distancia es **<=** que la **AcceptableDistance** y devolver el valor con el nodo **Return Node**.

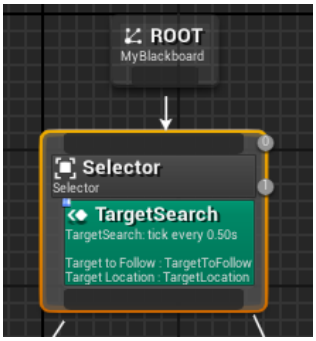


Ya tenemos nuestro decorador que devuelve true si el target está a menos distancia que el AcceptableRadius que se le configure.

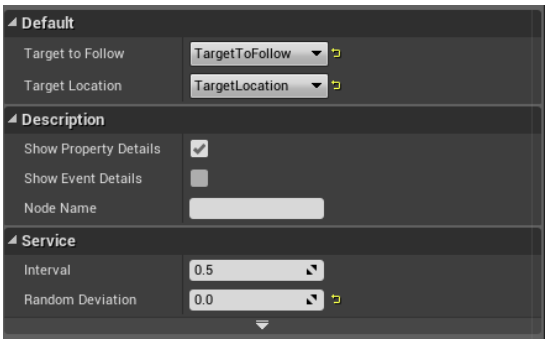
Behavior Tree

Ya solo queda montar el BT, sobre el esqueleto que montamos antes vamos a añadir y configurar los nuevos nodos que hemos creado.

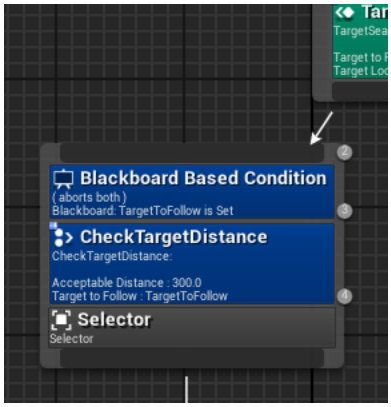
Al Primer **Selector** (debajo del Root) le añadimos el Service **TargetSearch**:



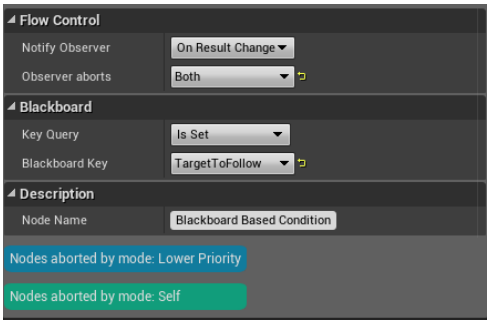
Al servicio **TargetSearch** le configuramos las variables de la blackboard que queremos usar **Target To Follow = TargetToFollow** y **Target Location = TargetLocation**. Configuramos además el intervalo de tick a **0.5** para que se haga el check cada medio segundo.



Al segundo **Selector** (abajo a la izquierda):



Le añadimos un **Decorator** del tipo **Blackboard Based Condition**, como condición seleccionamos **Is Set** y como variable a comprobar ponemos **TargetToFollow**. De esta forma este decorador devolverá true si tenemos un **Target To Follow**. También tenemos que configurar la política de control en la sección **Flow Control**, en este caso seleccionamos **On Result Change** y **Both**.



La política de control sirve para indicar al decorador que hacer en caso de que la condición que está comprobando cambie de valor y cuando hacer la comprobación:

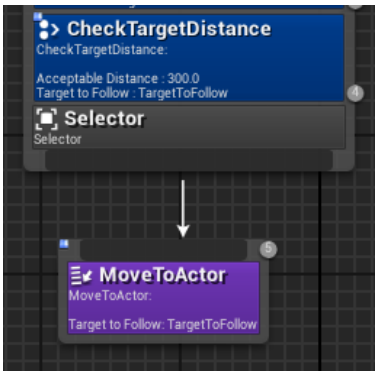
- ***Notify Observer***: Indica cuándo se va a evaluar de nuevo el observador: **On Result Change** es cuando cambie el resultado; **On Value Change** cuando el valor se cambie, con un Set por ejemplo.

- ***Observer Aborts***: Indica al nodo qué hacer cuando se aborta por el fallo del decorador. **none** no hace nada, **self** se aborta a sí mismo y todos los nodos que tenga por debajo, **lower priority** aborta cualquier nodo a la derecha de este nodo y por último **both** que combina **self** y **lower priority**.

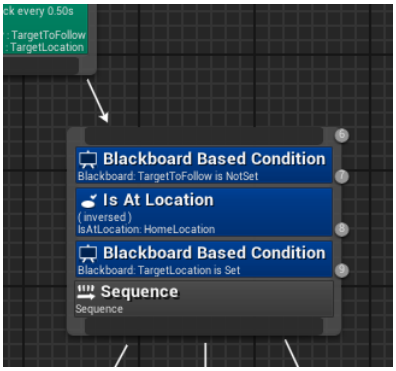
También le añadimos un decorador del tipo **CheckTargetDistance,** que hemos creamos anteriormente, y le configuramos una **AcceptableDistance = 300** y **Target To Follow = TargetToFollow**.

Es esta rama entraremos por tanto cuando tengamos target y esté a menos de 300 cm de distancia.

Debajo de este selector vamos a añadir la tarea **MoveToActor,** que creamos anteriormente, para movernos hacia el **TargetToFollow** si se cumplen los dos decoradores anteriores: tenemos target y está a una distancia menor de 300 cm. Debemos configurar a la tarea el **Target to Follow = TargetToFollow**.



Ya tenemos toda la rama izquierda del árbol que queremos construir, ahora vamos a añadir decoradores a la **Sequence** de la derecha:



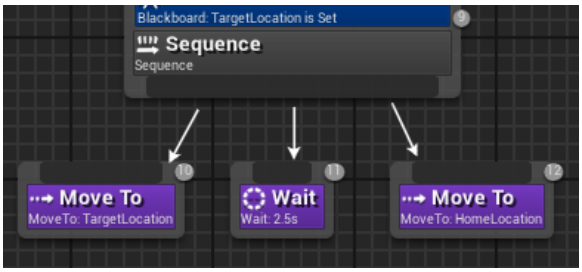
Añadimos un decorador **Blackboard Based Condition,** también apuntando a **TargetToFollow**, pero en este caso con **Is NotSet** para que entre por esta rama cuando **no** tenemos target.

Un segundo decorador tipo **Is At Location** apuntando a la variable **HomeLocation** para comprobar que no estemos ya en casa (el punto de spawn), por lo que debemos seleccionar Inverse Condition.

Por último, otro decorador del tipo **Blackboard Based Condition** pero apuntando a **TargetLocation** para comprobar si tenemos una posición donde buscar al target (última posición conocida que almacenada el servicio **TargetSearch**).

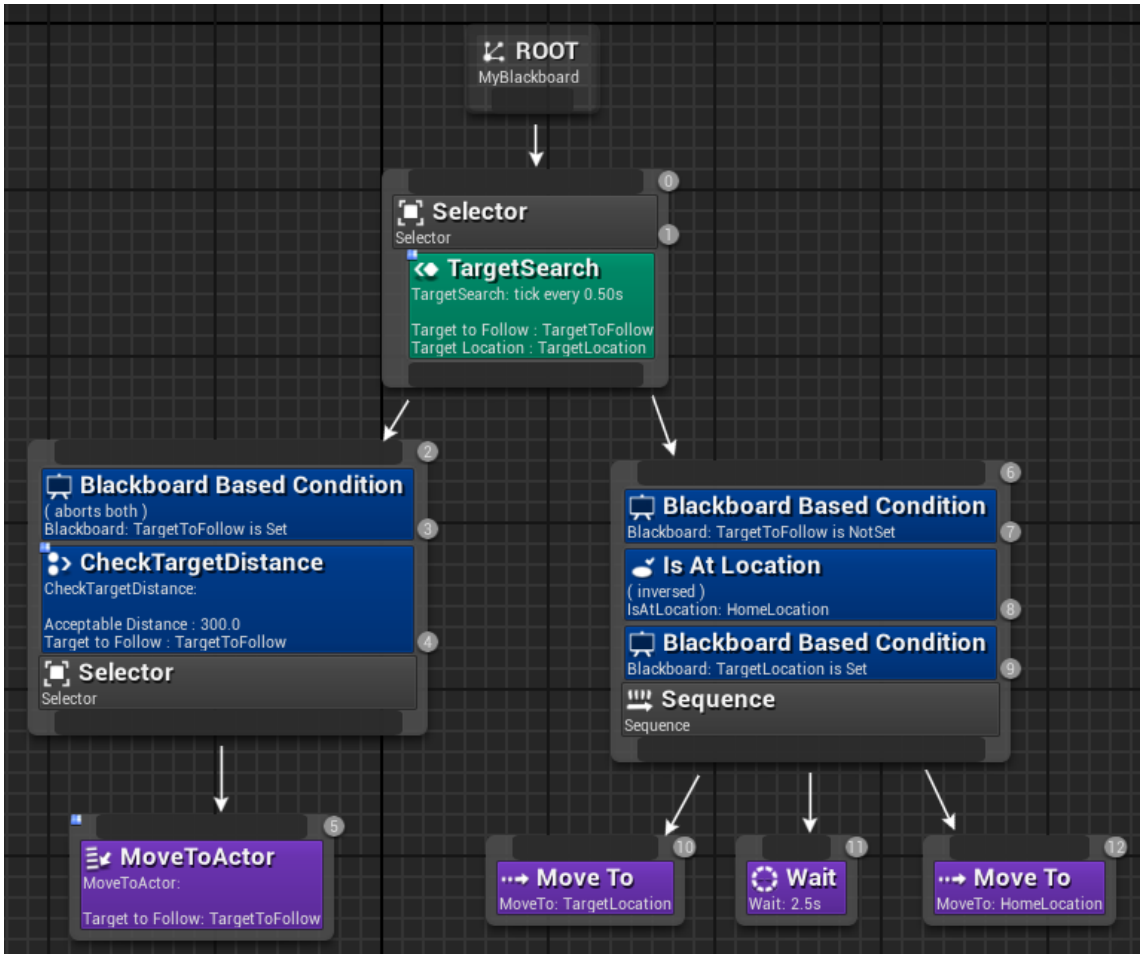
En esta rama entraremos cuando no tengamos target, no estemos en casa y tengamos un punto donde buscar al target.

Debajo de esta secuencia vamos a realizar 3 pasos: primero buscamos al jugador en la última posición conocida, después esperamos 2.5 segundos y finalmente volvemos a casa.



Para esto podemos usar los nodos que ya nos ofrece UE4, un **MoveTo TargetLocation** seguido de un **Wait** de 2.5s y otro **MoveTo HomeLocation**.

Nuestro árbol debería quedar así:

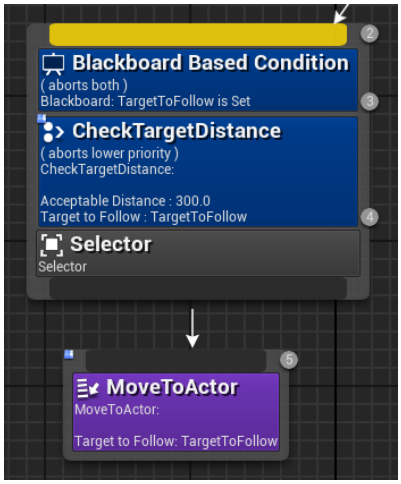


¡Vamos a probarlo!

IMPORTANTE: Para que funcione el depurador de BT tiene que estar abierto cuando se lanza la ejecución; no es como los blueprints que se abren cuando salta un breakpoint.

¡Parece que todo funciona como esperábamos! Pero si jugamos un poco con el comportamiento podemos ver que una vez que comienza a volver hacia la casa no reacciona de nuevo si detecta al jugador a menos de 300 cm de distancia. ¿Cómo podemos arreglarlo? ¿Por qué pasa esto?

Lo que pasa es que el decorador **CheckTargetDistance** está configurado con **ObserverAborts = none;** esto quiere decir que, aunque se cumpla la condición, no va a abortar los nodos que tiene a su derecha. Vamos a probar a poner este decorador como **Lower Priority** para que, si se cumple, aborte los nodos que estén a su derecha; no lo ponemos como both (aunque en este caso daría igual) porque no queremos que aborte el nodo que tiene debajo, cuando se aborta un nodo que está haciendo un move éste no se aborta hasta que no lo paramos con un **Stop Movement**.



Parte 3: Patrulla en BehaviorTree con C++

El objetivo es tener un Character controlado por la IA que recorra una ruta de waypoints

Lo que crearemos en este ejercicio es un enemigo cuyo AIController, creado en C++, lanzará un BehaviorTree que utilice Tasks y Servicios creados en C++. Crearemos también en C++ una clase WayPoints que será la que determinará la ruta de patrulla de dicho enemigo.

Para esta parte de la práctica vamos a crear un nuevo proyecto, similar al anterior, pero esta vez eligiendo que sea una plantilla de C++. Elegiremos por tanto la plantilla TopDownCharacter en C++. Automáticamente se nos creará todo el código asociado al proyecto, así como el archivo de la solución .sln que nos permitirá abrir el VisualStudio para editar nuestro código C++. Podremos compilar ya nuestro proyecto y éste se encargará automáticamente de actualizar el Editor de Unreal con nuestros cambios cada vez que compilemos.

De vuelta en el Editor de Unreal vamos a, al igual que en el ejemplo anterior, crear los elementos que necesitaremos para este ejercicio:

1) Creamos la NavMesh como en ejercicios anteriores.

2) Creamos nuestro **Character** AI_Character como hicimos el primer día. Vamos a instanciarlo en el nivel, arrastrando y soltando.

3) Creamos un **Behavior Tree;** para ello hacemos clic derecho en la carpeta de los blueprints del **Content Browser** y lo creamos seleccionándolo en el menú **Create Advanced Assets/Artificial Intelligence/Behavior Tree**. Lo llamamos MyBehaviorTree.

4) Creamos la Blackboard que el igual que el Behavior Tree está en **Create Advanced Assets/Artificial Intelligence/Blackboard**. La llamamos MyBlackboard.

Ahora crearemos una serie de clases de C++, el **AIcontroller y el Waypoint**.

- **El AIController** lo crearemos como clase BluePrint hija de nuestra clase de C++ y desde él lanzaremos la ejecución del Behavior Tree.
    - Para crear la clase de C++ que será el padre vamos a: **File/Add New C++ Class** y seleccionamos que herede de AIController. La llamaremos AIEnemyCppController. Al darle a "Create", Unreal creará nuestra clase y la dará de alta en el proyecto en el VisualStudio. Si abrimos la solución del proyecto podremos ver los dos ficheros que se nos han creado automáticamente: AIEnemyCppController.h y AIEnemyCppController.cpp. Compilamos el código; una vez que haya compilado ya

podremos utilizar esta clase desde el Editor de Unreal porque, como ya hemos comentado, se recarga automáticamente todo lo que hayamos añadido desde el VisualStudio.

- Lo que queremos ahora es crear una nueva clase **Blueprint**, como hemos hecho en otras ocasiones, pero esta vez esta clase será hija de nuestra clase de C++ recién creada: AIEnemyCppController. Para ello vamos al **ContentBrowser** y hacemos **AddNew** y clicamos en **NewBluePrintClass**; seleccionamos AIEnemyCppController. Lo llamaremos AIEnemyCppControllerBlueprint; con esto tendremos nuestro controlador hijo de la clase de C++, pero editable desde BluePrint.

- Ya podemos asignarle este controlador a la instancia del **AICharacter** que creamos antes en el nivel. Como siempre, seleccionamos el controlador de "controlará" a dicho personaje en sus propiedades:
  AIController → AIEnemyCppControllerBluePrint

Si compilamos, guardamos y ejecutamos desde Unreal podremos ver que aún no hace nada el personaje, pero de momento vamos bien.

- Vamos a invocar el **BehaviorTree** que hemos creado desde el AIEnemyCppControllerBlueprint; editamos éste y en el EventGraph vamos a añadir el nodo que lanza el BehaviorTree tras el **EventBeginPlay**. El nodo es **RunBehaviorTree** y en él debemos seleccionar el árbol que queremos que ejecute: MyBehaviorTree.

- **El WayPoint:** lo crearemos como clase exclusivamente de C++, para ver que podemos instanciar en el nivel también objetos de una clase exclusivamente de C++, sin necesidad de que sea BluePrint.

  - Para ello, igual que hicimos antes con el AIController, vamos a **File/Add new C++** class y seleccionaremos **TargetPoint** como clase padre de nuestra nueva clase, a la que llamaremos AIEnemyTargetPoint.

  - Igual que antes se nos habrán añadido al proyecto de VisualStudio un par de nuevos archivos AIEnemyTargetPoint.h y AIEnemyTargetPoint.cpp. Vamos a añadir el atributo de clase: **Position**

```
UCLASS()
class BTSCONCODIGO_API ATargetPointCpp : public ATargetPoint
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Ejercicio")
    int32 m_iPosition;

};
```

- Gracias al macro **UPROPERTY** con el atributo **EditAnywhere** podemos ver **Position** desde el Editor y editar su valor. Con **BlueprintReadWrite** lo definimos para que también lo podamos manipular desde el Blueprint y Category representa un nombre de sección para mostrarlo en el Editor. Verás que el atributo **Position** sale en el panel de detalles del actor en una sección de nombre Ejercicio.

- Compilamos desde el VisualStudio y, cuando termine, ya podremos utilizar esta clase desde el editor de Unreal.

- Vamos a crear las instancias de los **Waypoints** en el nivel; para ello vamos a la pestaña Modes, seleccionando AllClasses; podremos buscar nuestra nueva clase, exclusivamente de C++, AIEnemyTargetPoint. Creamos 4 instancias, una por cada esquina del nivel. Modificaremos para cada una el atributo que hemos publicado, **Position**, seleccionando números del 0 al 3.

- Antes de ponernos con el **BehaviorTree** y añadir los nuevos **Task** vamos a crear una nueva **Key** en la **Blackboard** para llevar la cuenta del índice de **Waypoint** por el que vamos. Será de tipo **Int** y lo llamaremos ***TargetPointIndex***. También crearemos una nueva **Key** de tipo **Vector** para almacenar la posición del Waypoint al que debe ir la entidad y la llamaremos **TargetPointPosition**.

- Vamos ahora sí a crear el **Task** necesario para que nuestra IA haga su patrulla por el nivel recorriendo los **Waypoints** que hemos colocado.

  - La función que llamaremos desde este **Task** la vamos a colocar en nuestro AIController y será **UpdateNextTargetPoint**. Dicha función la llamaremos desde un **Task** creado en C++, aunque la vamos a definir de manera que también se pudiera invocar desde un Blueprint. En AIEnemyController.h incluiremos la definición de la función:

```
public:

    UFUNCTION(BlueprintCallable, Category = "Ejercicio")
    void UpdateNextTargetPoint();
```

- En el .cpp vamos a crear la implementación de esta función:

  1. Si el enemigo ha llegado al último Waypoint, empezamos por el primero. Para determinar esto debemos ver cuál es el índice que tenemos almacenado en la Blackboard. Para acceder a ésta accederemos al **BlackBoardComponent** que tiene el **BrainComponent**, una variable de la clase padre de nuestra clase: AIController.

  2. Recorremos todos los Waypoints para encontrar el que tenga en su variable **Position** el mismo valor que el del índice actual que tenemos en la Blackboard. Para recorrer todos los elementos de un tipo usaremos el iterador **TActorIterator**, en el que mediante un template determinaremos de que clase serán los elementos que queramos recorrer.

  3. Almacenamos la posición de dicho Waypoint en la variable de la Blackboard **TargetPointPosition**

  4. Incrementamos el índice en la Blackboard que indica por qué Waypoint vamos

```cpp
void AAIEnemyCppController::UpdateNextTargetPoint()
{
    //Obtiene la referencia al BlackboardComponent del AIController
    UBlackboardComponent* pBlackboardComponent = BrainComponent->GetBlackboardComponent();

    //Guarda en TargetPointIndex el valor que tiene el Blackboard en el KEY TargetPointIndex
    //Este numero representa el orden en el que se mueve el enemigo por los TargetPoint del nivel
    int32 iTargetPointIndex = pBlackboardComponent->GetValueAsInt("TargetPointIndex");

    // TODO: Meter en la Blackboard un MAX_TARGET_POINTS o mejor, sacar este valor de iterar los que haya en el nivel...
    //Como solo tenemos 4 TargetPoint, cuando ya esté en el último, que lo reinicie al primero.
    if (iTargetPointIndex >= 4)
    {
        //Pone en 0 el valor del KEY TargetPointIndex del Blackboard
        iTargetPointIndex = 0;
        pBlackboardComponent->SetValueAsInt("TargetPointIndex", iTargetPointIndex);
    }

    //Iteramos por todos los AAIEnemyTargetPointCpp que hay en el nivel
    for (TActorIterator<ATargetPointCpp> It(GetWorld()); It; ++It)
    {
        //Obtenemos el TargetPoint actualmente en el ciclo
        ATargetPointCpp* pTargetPoint = *It;

        //Si el TargetPointIndex del BlackBoard es igual al valor del atributo Position del AAIEnemyTargetPointCpp
        //Este es el siguiente punto al que tiene que moverse el NPC por lo que setteamos el KEY TargetPointPosition con la posicion de ese Actor
        if (iTargetPointIndex == pTargetPoint->Position)
        {
            //Setteamos el KEY TargetPointPosition con la posicion de ese TargetPoint en el nivel y detenemos el ciclo con el break;
            pBlackboardComponent->SetValueAsVector("TargetPointPosition", pTargetPoint->GetActorLocation());
            break;
        }
    }

    //Por último, incrementamos el valor de TargetPointIndex del Blackboard
    pBlackboardComponent->SetValueAsInt("TargetPointIndex", (iTargetPointIndex + 1));

}
```

Si compilamos el código veremos que tenemos errores, muchos errores:

E0393	no se permite un puntero a un tipo de clase incompleta
E0393	no se permite un puntero a un tipo de clase incompleta
E0393	no se permite un puntero a un tipo de clase incompleta
E0864	TActorIterator no es una plantilla
E0020	el identificador "ATargetPointCpp" no está definido
E0020	el identificador "It" no está definido
E0020	el identificador "pTargetPoint" no está definido
E0393	no se permite un puntero a un tipo de clase incompleta
E0393	no se permite un puntero a un tipo de clase incompleta
E0393	no se permite un puntero a un tipo de clase incompleta
E0393	no se permite un puntero a un tipo de clase incompleta
E0393	no se permite un puntero a un tipo de clase incompleta
E0393	no se permite un puntero a un tipo de clase incompleta
E0393	no se permite un puntero a un tipo de clase incompleta
C2027	uso del tipo 'UBrainComponent' sin definir
C2227	el operando izquierdo de '->GetBlackboardComponent' debe señalar al tipo class/struct/union/generic
C2027	uso del tipo 'UBlackboardComponent' sin definir
C2227	el operando izquierdo de '->GetValueAsInt' debe señalar al tipo class/struct/union/generic
C2027	uso del tipo 'UBlackboardComponent' sin definir
C2227	el operando izquierdo de '->SetValueAsInt' debe señalar al tipo class/struct/union/generic
C2065	'TActorIterator': identificador no declarado
C2065	'ATargetPointCpp': identificador no declarado
C3861	'It': no se encontró el identificador
C2065	'It': identificador no declarado
C2065	'ATargetPointCpp': identificador no declarado
C2065	'pTargetPoint': identificador no declarado
C2065	'It': identificador no declarado
C2065	'pTargetPoint': identificador no declarado
C2227	el operando izquierdo de '->Position' debe señalar al tipo class/struct/union/generic
C2027	uso del tipo 'UBlackboardComponent' sin definir
C2227	el operando izquierdo de '->SetValueAsVector' debe señalar al tipo class/struct/union/generic
C2065	'pTargetPoint': identificador no declarado
C2227	el operando izquierdo de '->GetActorLocation' debe señalar al tipo class/struct/union/generic
C2027	uso del tipo 'UBlackboardComponent' sin definir
C2227	el operando izquierdo de '->SetValueAsInt' debe señalar al tipo class/struct/union/generic
C2027	uso del tipo 'UBrainComponent' sin definir
C2227	el operando izquierdo de '->GetBlackboardComponent' debe señalar al tipo class/struct/union/generic
C2027	uso del tipo 'UBlackboardComponent' sin definir
C2227	el operando izquierdo de '->SetValueAsObject' debe señalar al tipo class/struct/union/generic
C2027	uso del tipo 'UBlackboardComponent' sin definir
C2227	el operando izquierdo de '->SetValueAsObject' debe señalar al tipo class/struct/union/generic
C2027	uso del tipo 'UBrainComponent' sin definir
C2227	el operando izquierdo de '->GetBlackboardComponent' debe señalar al tipo class/struct/union/generic
C2027	uso del tipo 'UBlackboardComponent' sin definir
C2227	el operando izquierdo de '->GetValueAsObject' debe señalar al tipo class/struct/union/generic

Son simplemente errores de compilación por no haber incluido los archivos .h necesarios de los que hemos utilizado su funcionalidad en este caso. Probad a resolver por vuestra cuenta estos errores. Algunos serán bastante obvios, pero otros tendréis que ver en qué fichero .h se encuentran las definiciones de los métodos utilizados.

Aquí os dejo los includes necesarios, pero insisto, antes de copiarlos sin pensar intentad resolver todos los errores de compilación por vuestra cuenta.

```cpp
#include "BTsConCodigo.h"
#include "AIEnemyCppController.h"
#include "BrainComponent.h"
#include "BehaviorTree/BlackboardComponent.h"
#include "TargetPointCpp.h"
#include "Runtime/Engine/Public/EngineUtils.h"
```

Como veis los dos primeros ya los teníamos: el primero es el del nombre del proyecto cuando comenzamos la práctica. El segundo la definición de nuestra clase AIEnemyCppController. Los que vienen a continuación sí son los que debemos añadir para continuar. Ahora ya podremos compilar el código sin errores y poder volver al Editor de Unreal.

Este método que hemos creado en el controlador lo queremos llamar desde el **BehaviorTree**, por lo que necesitamos crear un **Task** que será el encargado de invocar esta función.

Como siempre desde el editor hacemos **File/New C++ class** y heredaremos de **BTTaskNode**. Llamaremos a nuestra nueva clase **UpdateNextTargetPointBTTaskNode**. Al crearla nos añade automáticamente los archivos .h y .cpp.

En el .h necesitaremos definir las funciones:

```cpp
/* Se llama al iniciar este Task, tiene que retornar Succeeded, Failed o InProgress */
virtual EBTNodeResult::Type ExecuteTask(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory);

/** Permite definir una descripción para este Task. Este texto se ve en el Nodo al agregarlo al Behavior Tree */
virtual FString GetStaticDescription() const override;
```

La función **ExecuteTask** será la que se ejecute el iniciar el **Task** y se encargará de informar del resultado de su ejecución para que el **BehaviorTree** se ejecute correctamente.

La función **GetStaticDescription** se usará desde el Editor para darnos información acerca del **Task** cuando lo añadamos.

En el .cpp meteremos las implementaciones de ambas funciones; en este caso serán muy sencillas: En el **ExecuteTask** accederemos al **AIController** e invocaremos la función que creamos antes: **UpdateNextTargetPoint** y devolveremos **Success** para que el padre de este nodo en el **BehaviorTree** pueda continuar con su ejecución.

En el caso de la descripción simplemente devolveremos el texto informativo del **Task**.

```
/* Se llama al iniciar este Task, tiene que retornar Succeeded, Failed o InProgress */
EBTNodeResult::Type UUpdateNextTargetPointBTTaskNode::ExecuteTask(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory)
{
    //Obtenemos la referencia al AIEnemyController
    AAIEnemyCppController* AIEnemyController = Cast<AAIEnemyCppController>(OwnerComp.GetOwner());

    //Llamamos al método UpdateNextTargetPoint que tiene la lógica para seleccionar el siguiente TargetPoint
    AIEnemyController->UpdateNextTargetPoint();

    //Finalmente retornamos Succeeded
    return EBTNodeResult::Succeeded;
}

/** Permite definir una descripción para este Task. Este texto se ve en el Nodo al agregarlo al Behavior Tree */
FString UUpdateNextTargetPointBTTaskNode::GetStaticDescription() const
{
    return TEXT("Actualiza el siguiente punto en el recorrido");
}
```

En esta ocasión nos bastará con incluir el .h del AIEnemyCppController donde tenemos la definición de la función que vamos a invocar.

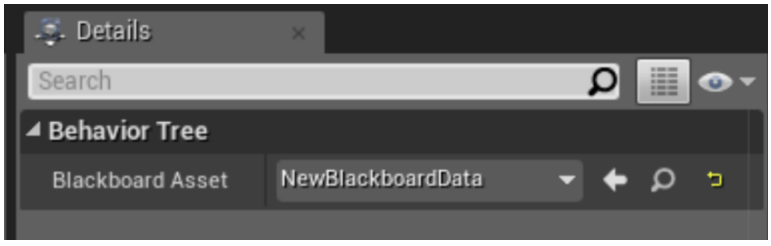
Compilamos y, si todo está ok, ya podremos utilizar esta **Task** desde el Editor de Unreal.

Abrimos el **BehaviorTree** e introduciremos la funcionalidad necesaria que hemos creado previamente para que nuestro personaje ejecute su ruta. Necesitaremos un nodo Secuencia que ejecute, uno detrás de otro los siguientes nodos: ElegirWaypoint, ir hasta él, esperar. Y así una y otra vez para crear la patrulla por los Waypoints.

El primer nodo que añadiremos al árbol será el **Task** que acabamos de crear. Lo encontraremos en **Task/UpdateNextTargetPointBTTaskNode.**

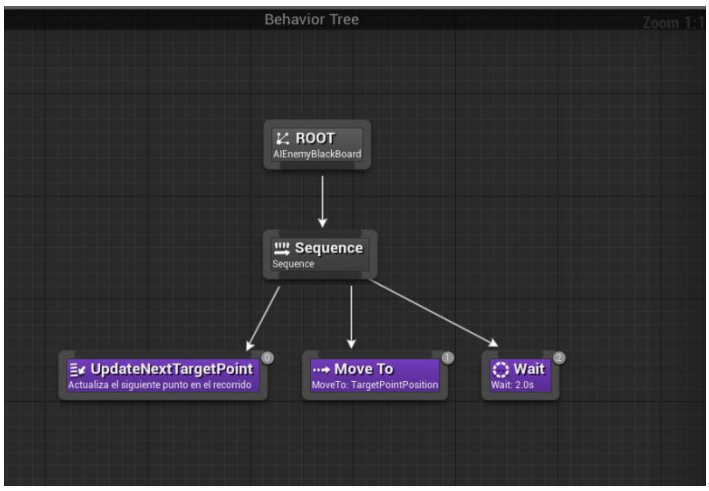
A continuación, en la secuencia crearemos un nodo **MoveTo** cuyo parámetro será la variable de la **BlackBoard** que creamos y que se llama **TargetPointPosition**. Si en el selector no apreciaran nuestras variables de la Blackboard eso significa que no tenemos asociados árbol y pizarra.

Para hacer esto hacemos clics en una zona vacía del árbol, y en las propiedades, veremos que podemos seleccionar una Blackboard; elegiremos la blackboard que creamos al comienzo del ejercicio.



Ahora ya sí podremos elegir en el nodo **MoveTo** la variable **TargetPointPosition**. Este nodo devolverá **Success** cuando llegue al objetivo asignado, con lo que nuestra secuencia continuará.

Por último, creamos un nodo **Wait** con un par de segundos, por ejemplo; cuando este nodo acabe el nodo secuencia devolverá **Success**. Al llegar a la raíz del árbol, en el siguiente **Tick** se volverá a comenzar por el principio, con lo que se volverá a ejecutar nuestra **Task**, se elegirá el siguiente Waypoint y reanudará la patrulla.



Si ejecutamos podremos ver que el enemigo hace su patrulla pero nos ignora, ya que no hemos implementado nada más que la patrulla, pero aún nada de “detección”.

Parte 4: Ampliando el árbol en C++: Chase

El objetivo es tener un Character controlado por la IA que recorra una ruta de waypoints y cuando "detecte" a un jugador lo persiga

Vamos a añadir al ejercicio anterior la capacidad a nuestra inteligencia artificial de detectarnos y perseguirnos. Para ello crearemos un **Servicio** y un nuevo **Task** en C++ que se encarguen de ello y se los añadiremos a nuestro **BehaviorTree**.

Lo primero que haremos será implementar en nuestro **AIEnemyCppController** una función que llamaremos desde el **Servicio** para comprobar si hay algún jugador en las cercanías de nuestro agente como hicimos en la parte 2 pero un poco más simplificado.

Primero vamos a crear una nueva **Key** en la **Blackboard** donde almacenaremos al player detectado para poder ir a por él en el árbol. Será una **Key** de tipo **Object** y la llamaremos **TargetActorToFollow**.

A continuación, añadimos en el archivo AIEnemyCppController .h la definición de esta función:

```
UFUNCTION(BlueprintCallable, Category = "Ejercicio")
    void CheckNearbyEnemy();
```

Ahora la implementamos en AIEnemyCppController.cpp: Básicamente vamos a lanzar un testeo de esfera, mediante la función **SphereTraceMultiForObjects**, desde la posición de nuestro agente para comprobar si colisiona con algún player. En caso de encontrar alguno escribiremos una referencia al mismo en la variable de la BlackBoard **TargetActorToFollow**.

Como os dije anteriormente os recomiendo intentar escribir este código por vuestra cuenta, pensando qué haría falta para hacer esta detección de esfera, cómo escribir en la blackboard el character encontrado, etc. A continuación, os dejo el código ya resuelto.

```cpp
void AAIEnemyCppController::CheckNearbyEnemy()
{
    //Obtenemos la referecia al Pawn del NPC
    APawn *pPawn = GetPawn();

    //Guardamos en MultiSphereStart la posición del NPC
    FVector MultiSphereStart = pPawn->GetActorLocation();

    //Creamos un nuevo vector a partir de la posición del NPC pero con un incremento en la Z de 15 unidades.
    //Estos dos valores serán los puntos de inicio y fin de la llamada a MultiSphereTraceForObjects
    FVector MultiSphereEnd = MultiSphereStart + FVector(0, 0, 15.0f);

    //Creamos el array que pasaremos como el parámetro ObjectTypes del MultiSphereTraceForObjects y define los tipos de objetos a tener en cuenta
    TArray<TEnumAsByte<EObjectTypeQuery>> ObjectTypes;
    ObjectTypes.Add(UEngineTypes::ConvertToObjectType(ECC_Pawn));

    //Creamos el array de los actores a ignorar por el método, solamente con nuestro propio Pawn
    TArray<AActor*> ActorsToIgnore;
    ActorsToIgnore.Add(pPawn);

    //OutHits la usaremos como parámetro de salida. Como el método SphereTraceMultiForObjects recibe la referencia de este parámetro
    //al terminar la ejecución del método en este array se encuentran el array de FHitResult con los Hits que detecte el método.
    TArray<FHitResult> OutHits;

    bool bResult = UKismetSystemLibrary::SphereTraceMultiForObjects(GetWorld(),     //Referencia del Mundo
        MultiSphereStart,                                                           //Punto de Inicio de la recta que define la esfera
        MultiSphereEnd,                                                             //Punto de fin de la recta que define la esfera
        700,                                                                        //Radio de la esfera
        ObjectTypes,                                                                //Tipos de objetos a tener en cuenta en el proceso
        false,                                                                      //No queremos que se use el modo complejo
        ActorsToIgnore,                                                             //Los actores que se van a ignorar aunque esten dentro de la esfera
        EDrawDebugTrace::ForDuration,                                               //El tipo de Debug
        OutHits,                                                                    //Parámetro por referencia donde se guardarán los resultados
        true);                                                                      //si se ignora el propio objeto

    UBlackboardComponent* BlackboardComponent = BrainComponent->GetBlackboardComponent();

    //Si hay resultados en el Trace
    if (bResult)
    {
        //Recorremos todos los objetos dentro del OutHits
        for (int32 i = 0; i < OutHits.Num(); i++)
        {
            //Obtenemos el FHitResult actual
            FHitResult Hit = OutHits[i];

            //Obtenemos la referencia el Character
            ACharacter* pCharacter = UGameplayStatics::GetPlayerCharacter(GetWorld(), 0);

            //Si el Actor detectado en el Trace es igual al Character
            //setamos el KEY TargetActorToFollow del Blackboard con la referencia al Character
            if (Hit.GetActor() == pCharacter)
            {
                BlackboardComponent->SetValueAsObject("TargetActorToFollow", pCharacter);
                break;
            }
        }
    }
    else
    {
        // En caso de no detectar ninguna colisón establecemos a NULL TargetActorToFollow
        BlackboardComponent->SetValueAsObject("TargetActorToFollow", NULL);
    }
}
```

Como veis, simplemente se lanza el testeo y con los resultados se escribe en la Blackboard. Por tanto, necesitaremos implementar el **Servicio** que invoque esta función para que se pueda ejecutar en el árbol. Para poder llamar desde éste la función que acabamos de crear debemos compilar nuestro código sin errores.

Para crear el **Servicio** vamos a crear desde el Editor de Unreal una nueva clase de C++. Hacemos clics en **File/New C++ Class...** y heredamos de **BTService**. Llamamos a nuestra nueva clase **CheckNearbyEnemyBTService**.

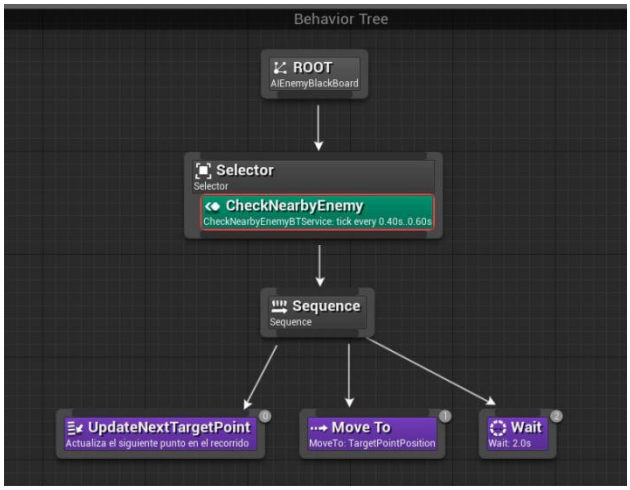
Como siempre se nos crean el .h y el .cpp en nuestro proyecto y vamos a editarlos desde el VisualStudio. Vamos a crear la definición de la función del **Servicio** que queremos sobrescribir, el **TickNode**, que será llamada cada Tick del árbol.

```
8   /**
9    *
10   */
11  UCLASS()
12  class BTSCONCODIGO_API UUCheckNearbyEnemyBTService : public UBTService
13  {
14      GENERATED_BODY()
15
16          /** update next tick interval
17          * this function should be considered as const (don't modify state of object) if node is not instanced! */
18          virtual void TickNode(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory, float DeltaSeconds) override;
19
20  };
21
```

Vamos al .cpp a implementar el **Servicio**. Lo único que haremos será invocar el **Tick** de la clase padre y a continuación hacer la llamada a nuestra función del AIEnemyCppController recién creada **CheckNearbyEnemy**.

```
#include "BrainComponent.h"
#include "BehaviorTree/BlackboardComponent.h"
#include "TargetPointCpp.h"
#include "Runtime/Engine/Public/EngineUtils.h"

void UUCheckNearbyEnemyBTService::TickNode(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory, float DeltaSeconds)
{
    //Llamamos a la implementación de la clase base primero
    Super::TickNode(OwnerComp, NodeMemory, DeltaSeconds);

    //Obtenemos la referencia del AIEnemyController
    AAIEnemyCppController* AIEnemyController = Cast<AAIEnemyCppController>(OwnerComp.GetOwner());

    //Llamamos al método CheckNearbyEnemy del AIEnemyController que tiene toda la lógica para determinar si el enemigo está cerca o no
    //y configurar el KEY correspondiente del Blackboard
    AIEnemyController->CheckNearbyEnemy();
}
```

Por supuesto deberemos asegurarnos de incluir todos los ficheros de definición necesarios para que nuestro código compile. Una vez que así sea ya podremos crear desde el **BehaviorTree** este **Servicio**. Modificamos el árbol que teníamos para introducir este **Servicio** de comprobación:



Guardamos y ejecutamos y veremos que nuestro enemigo continúa haciendo la ruta como siempre, pero veremos el **DebugRender** de los testeos de esfera y, si ejecutamos con la

ventana del BT abierta veremos que la variable TargetActorToFollow de la Blackboard es rellenada.

Pero de momento no nos hará ni caso, porque tenemos que implementar aún el **Task** que se encarga de ello. Primero vamos a crear la función en nuestro AIController que será llamada en dicho **Task** y será la encargada de mandar a nuestro agente a la caza del player. Creamos la definición de dicha función en el AIEnemyCppController.h:

```
    UFUNCTION(BlueprintCallable, Category = "Ejercicio")
        EPathFollowingRequestResult::Type MoveToEnemy();
};
```

En el .cpp implementamos dicha función. Simplemente obtendremos la referencia de la BlackBoard que nuestro **Servicio** rellenó e invocaremos a la función **MoveToActor** pasando como parámetro dicha referencia.

```
EPathFollowingRequestResult::Type AAIEnemyCppController::MoveToEnemy()
{
    //Obtenemos la referencia al BlackBoard
    UBlackboardComponent* BlackboardComponent = BrainComponent->GetBlackboardComponent();

    //Obtenemos la referencia al Actor guardado en el KEY TargetActorToFollow del BlackBoard
    AActor* HeroCharacterActor = Cast<AActor>(BlackboardComponent->GetValueAsObject("TargetActorToFollow"));

    //Iniciamos el proceso de perseguir al personaje
    EPathFollowingRequestResult::Type MoveToActorResult = MoveToActor(HeroCharacterActor);

    return MoveToActorResult;
}
```

Ahora ya sólo nos queda crear el **Task** que utilizará este método y terminar de construir el **BehaviorTree**. Para crear el **Task**, como siempre, desde el Editor de Unreal hacemos **File/New C++ Class** y heredamos de **BTTaskNode**. Llamaremos a esta nueva clase **MoveToEnemyBTTaskNode**. Veamos la definición del .h:

```
/**
 *
 */
UCLASS()
class BTSCONCODIGO_API UMoveToEnemyBTTaskNode : public UBTTaskNode
{
    GENERATED_BODY()

    /* Se llama al iniciar este Task, tiene que retornar Succeeded, Failed o InProgress */
    virtual EBTNodeResult::Type ExecuteTask(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory) override;

    /* Se llama constantemente. Es usado generalmente en Tasks que en el ExecuteTask retornan InProgress */
    virtual void TickTask(class UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory, float DeltaSeconds) override;

    /** @return Una descripción para este Task. Este texto se ve en el Nodo al agregarlo al Behavior Tree */
    virtual FString GetStaticDescription() const override;


};
```

Vemos que hemos definido más métodos a sobrescribir que en ocasiones anteriores. Como ya sabemos, los **Task** pueden devolver un valor de retorno inmediatamente, como en el caso del **Task** UpdateNextTargetPoint, en el que devolvíamos siempre **Succeeded** en su función **ExecuteTask**, que se llama al iniciar la ejecución del **Task**.

Pero también puede ser que un **Task** sea latente, es decir, se quede ejecutándose durante varios ciclos hasta que pueda dar **Suceeded** o **Failed**.

En estos casos además de la función **ExecuteTask** debemos hacer uso de la función **TickTask**, que se ejecutará cada **Tick**, donde podremos comprobar si nuestro **Task** ya ha terminado.

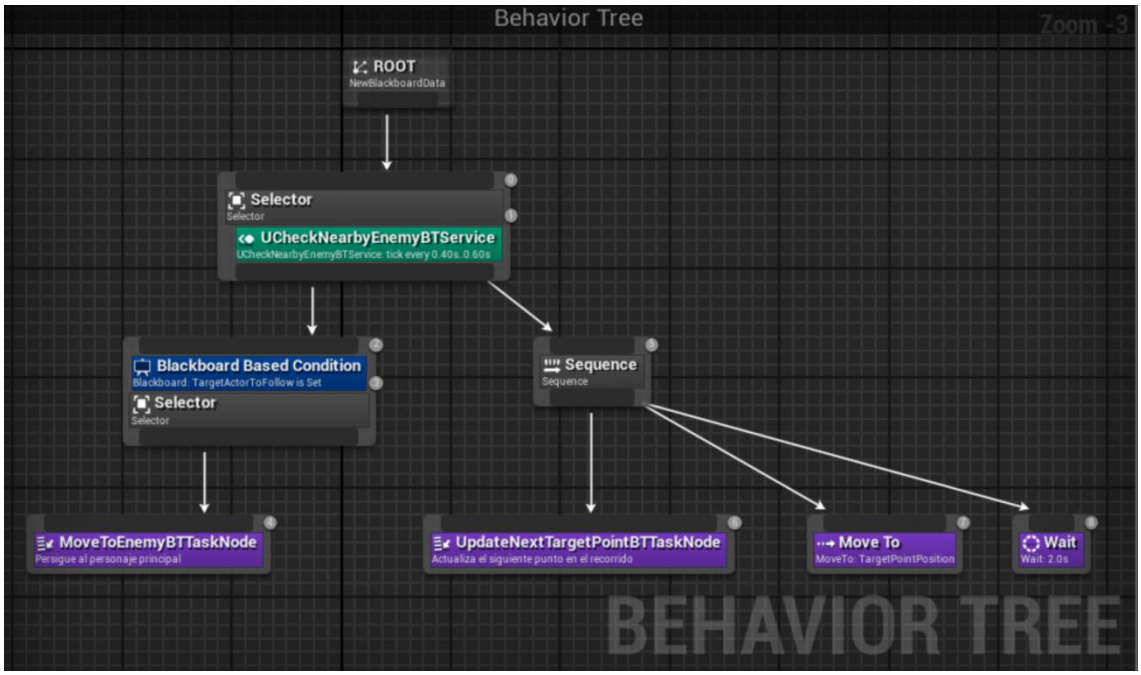
```
/* Se llama al iniciar este Task, tiene que retornar Succeeded, Failed o InProgress */
EBTNodeResult::Type UMoveToEnemyBTTaskNode::ExecuteTask(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory)
{
    return EBTNodeResult::InProgress;
}

/* Se llama constantemente. Es usado generalmente en Tasks que en el ExecuteTask retornan InProgress */
void UMoveToEnemyBTTaskNode::TickTask(class UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory, float DeltaSeconds)
{
    //Obtenemos la referencia al AIEnemyController
    AAIEnemyCppController* AIEnemyController = Cast<AAIEnemyCppController>(OwnerComp.GetOwner());

    //Llamamos al método MoveToEnemy del Controller y guardamos en MoveToActorResult el resultado
    EPathFollowingRequestResult::Type MoveToActorResult = AIEnemyController->MoveToEnemy();

    //Si ya se llega al objetivo se termina la ejecución el Task con FinishLatentTask y un resultado
    if (MoveToActorResult == EPathFollowingRequestResult::AlreadyAtGoal)
    {
        FinishLatentTask(OwnerComp, EBTNodeResult::Succeeded);
    }
}

/** @return Una descripción para este Task. Este texto se ve en el Nodo al agregarlo al Behavior Tree */
FString UMoveToEnemyBTTaskNode::GetStaticDescription() const
{
    return TEXT("Persigue al personaje principal");
}
```

Como vemos en la función que se llama al iniciar el Task, **ExecuteTask** vamos a devolver directamente el resultado **InProgress**. En este caso dejaremos que sea el **Tick** quien se encargue de mover a nuestro agente, aunque también podríamos hacer lo propio en esta función.

En el **TickTask** ejecutaremos la función **MoveToEnemy** de nuestro **AIEnemyCppController** que se encargará de todo. Por tanto, en esta función sólo deberemos comprobar si el resultado de dicha llamada ha sido **AlreadyAtGoal** lo que significará que nuestro **Task** puede terminar porque hemos alcanzado al player. En este caso llamaremos a la función **FinishLatentTask**, terminando la ejecución.

Compilamos el código y, cuando todo este ok, volveremos al Editor de Unreal para terminar de montar el **BehaviorTree**.

Vamos a añadir el **Task** que hemos creado desde C++, **MoveToEnemy**. Lo que queremos es que nuestro árbol, mediante el Servicio antes añadido, pueda rellenar la variable de la **Blackboard TargetActorToFollow**, con lo que si comprobamos que ha sido rellenada invoquemos el **Task** que lo persigue, pero en caso contrario que haga su patrulla. Añadiremos un selector que tenga un decorador que compruebe dicha variable de la Blackboard e invoque al **Task**.



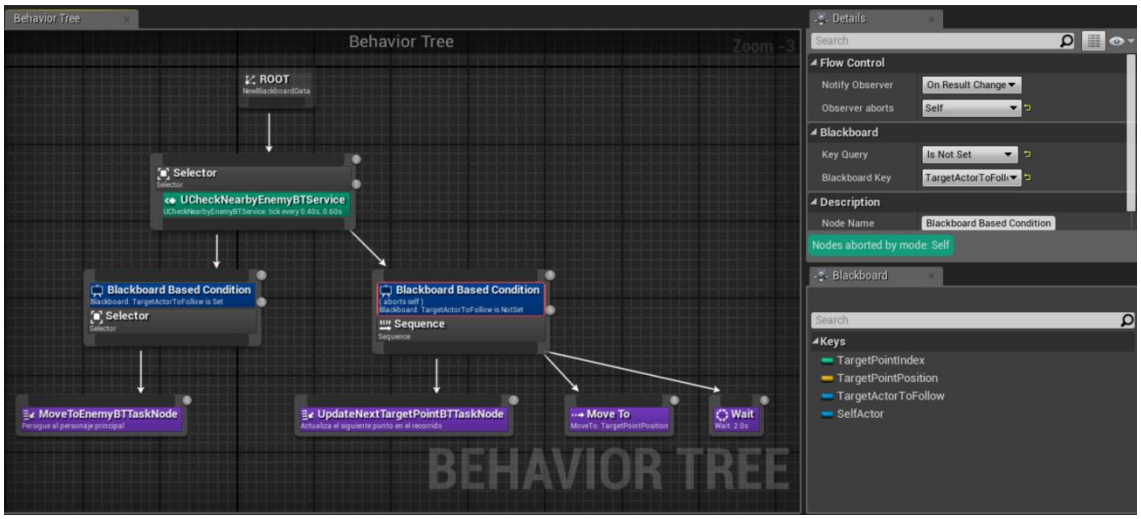
El **decorador** del **Selector** de la izquierda será de tipo Blackboard (que después aparecerá como **Blackboard Based Condition**) y lo que comprobaremos será la variable **TargetActorToFollow**. En caso de que esta condición se cumpla entonces se ejecutará el **Task MoveToEnemy** persiguiendo entonces al player.

Guardamos y ejecutamos; Veremos cómo nuestro agente comienza a hacer su ruta por el nivel lanzado los testeos de esfera desde el Servicio. En el momento que nos detecte vendrá a por nosotros... o no...

¿Qué está ocurriendo? Si como jugador, esperamos al enemigo cerca de un Waypoint de su ruta, cuando vaya a reanudar la marcha entonces sí vendrá a por nosotros, pero en cambio, si nos detecta a mitad de camino entre una esquina y la siguiente, aunque sí que nos esté detectando el Servicio (podemos ver en la ventana de **BehaviorTrees** que el valor de la **Key** de la **Blackboard** de **TargetActorToFollow** cambia al detectarnos) no interrumpirá su camino para venir a por nosotros...

¿Qué está ocurriendo? Simplemente que no se aborta en ese caso la rama derecha del árbol, la que le mandó hacer un **MoveTo** al siguiente **Waypoint**, por lo que hasta que no llegue al siguiente y haga su **Wait** no terminará la rama derecha del árbol y por tanto no podrá pasar la ejecución hacia arriba del árbol y, al llegar de nuevo al **Root** pasar a ejecutar la rama izquierda...

¿Cómo podemos resolver este problema? Debemos comprobar en la **Secuencia** de la rama derecha también la condición de si la variable **TargetActorToFollow NO** está “seteada”, lo que es equivalente a decir “NO nos ha detectado”. Probad a hacedlo por vuestra cuenta antes de ver la siguiente imagen, que muestra cómo se ha de configurar para que funcione correctamente.



Como veis en la imagen, hemos añadido la condición negada en el nodo secuencia de la rama derecha, es decir, comprobamos que la **Key** de la **Blackboard TargetActorToFollow NO** está “seteada”; pero no sólo eso, fijaros que en el campo **ObserverAborts** hemos elegido el valor **Self**, donde antes había **None**, de manera que esta rama pueda ser abortada en el momento que cambie el resultado de la condición.

Guardad y ejecutad y veréis que, ahora sí, el agente se lanza a por el jugador en el mismo momento de la detección, sin tener que esperar a que ningún otro proceso termine.

Como posible mejora podríamos añadir, como en la parte 2, el lanzamiento de un rayo para comprobar que haya visibilidad con el player antes de lanzar al agente a la caza.

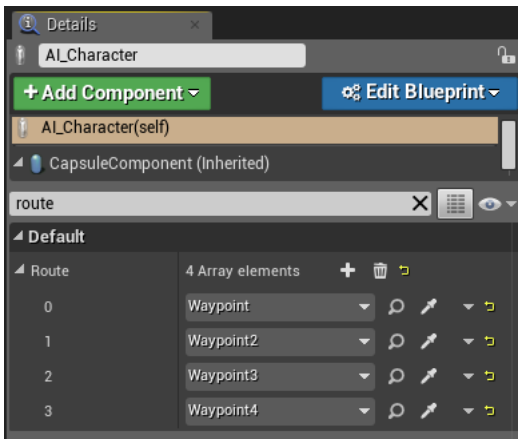
Parte 5: Ampliando la parte 2: Patrol and Chase en BluePrint → Ampliar también la parte 3 y 4 de C++

Vamos a ampliar ahora lo implementado en las partes anteriores; como vimos en la parte 2 teníamos un nodo casa y montamos un chase mediante un testeo de esferas primero para comprobar la "verdadera" visibilidad a continuación con un rayo. En las partes 3 y 4 hemos montado en C++ una patrulla por el nivel y una persecución sólo con la detección de esfera. Vamos a probar a añadir a cada proyecto la parte de funcionalidad del otro que no tenga implementada. En este caso explicaremos el paso a paso de añadir las rutas a la parte 2 sobre Blueprint, pero la parte de añadir al proyecto de C++ el testeo de visibilidad de rayo sería cosa vuestra ;) Os propongo ir añadiendo y modificando el proyecto de C++ con lo aprendido en el de Blueprint, como haríamos si un diseñador nos trae un prototipo que debemos trasladar de Blueprint a C++.

Ampliación de parte 2 en Blueprint:

En la parte 2 se propusieron una serie de mejoras; de todas las mejoras propuestas hemos elegido "En vez de dejar parado al AICharacter crear unas rutas que pueda seguir mientras no ve al jugador para defender una zona más amplia ".

Al igual que la Practica 1 vamos a crear un objeto de blueprint que represente nuestros waypoints (usando de base el **Target Point**) los llamaremos **Waypoints**, pero en este caso no vamos a crearle ninguna variable, lo que vamos a hacer es editar nuestro **AICharacter** para añadirle una variable array del tipo Waypoint donde le configuraremos la ruta. Ponemos unos cuantos Waypoints en el mapa y configuramos el AICharacter que teníamos ya puesto. Quedaría algo así:



Como hemos visto en la ejecución del BT que hemos montado el nodo **Move To – TargetToFollow** se ejecuta mientras se cumpla que tenemos target y que este está a menos de 300, cuando está a más de 300 como todavía tenemos target no se ejecuta el lado derecho del árbol que controla el movimiento a la última posición del target y después a la casa. Por lo

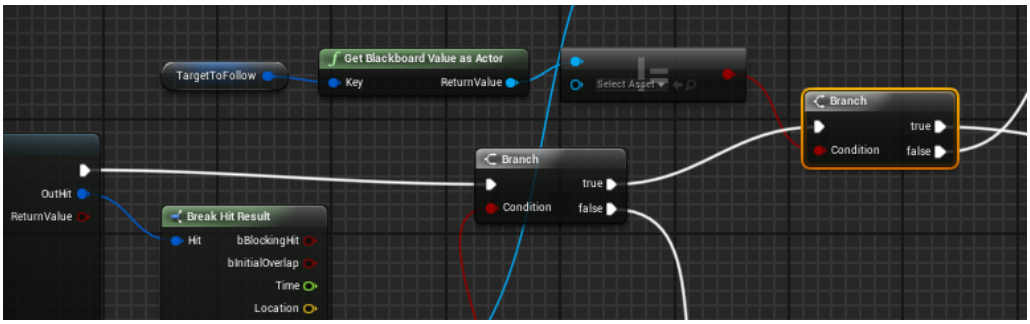
tanto, hasta que no perdemos al target nadie hace un **Move To** nuevo y no se aborta el **Move To – TargetFollow**.

El problema que surge ahora es que nuestro movimiento por defecto sí que implicar hacer **Move To** por lo que si planteamos el árbol de la misma forma no conseguiríamos el resultado deseado. ¿Cuál es la idea? Pues una de las muchísimas opciones que hay, seguro que las hay mejores y peores, es subir la comprobación de distancia al **Service – TargetSearch** de forma que sea el que tenga la lógica siguiente:

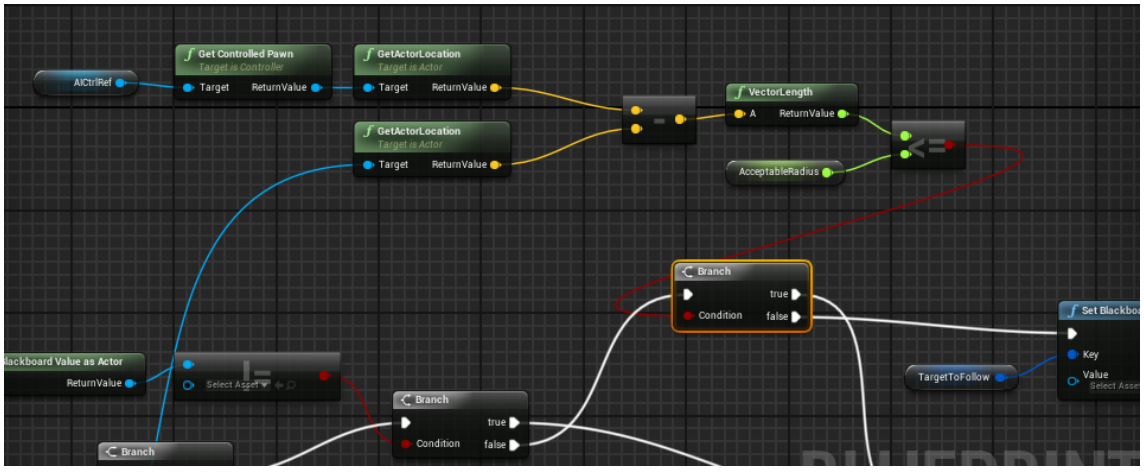
1. Si no tengo target lo busco y lo encuentro solo si está a menos de X distancia y tengo línea de visión con él.
2. Si ya tengo target no lo pierdo hasta que no tenga línea de visión con él.

De esta forma mirando el **TargetToFollow** sabremos si de verdad le tenemos que seguir o no.

¿Cómo lo hacemos? Pues tenemos que modificar el **Service – TargetSearch** para que cuando encuentra un hit valido mire si ya tenía target, es decir que **TargetToFollow** sea != NULL:



Y con un **Branch** si ya teníamos target lo mantenemos y si no teníamos tenemos que comprobar la distancia (publicamos la variable **AcceptableDistance** como hicimos en el decorador), tenemos que comparar la posición del **HitActor** con la de nuestro **Pawn**:



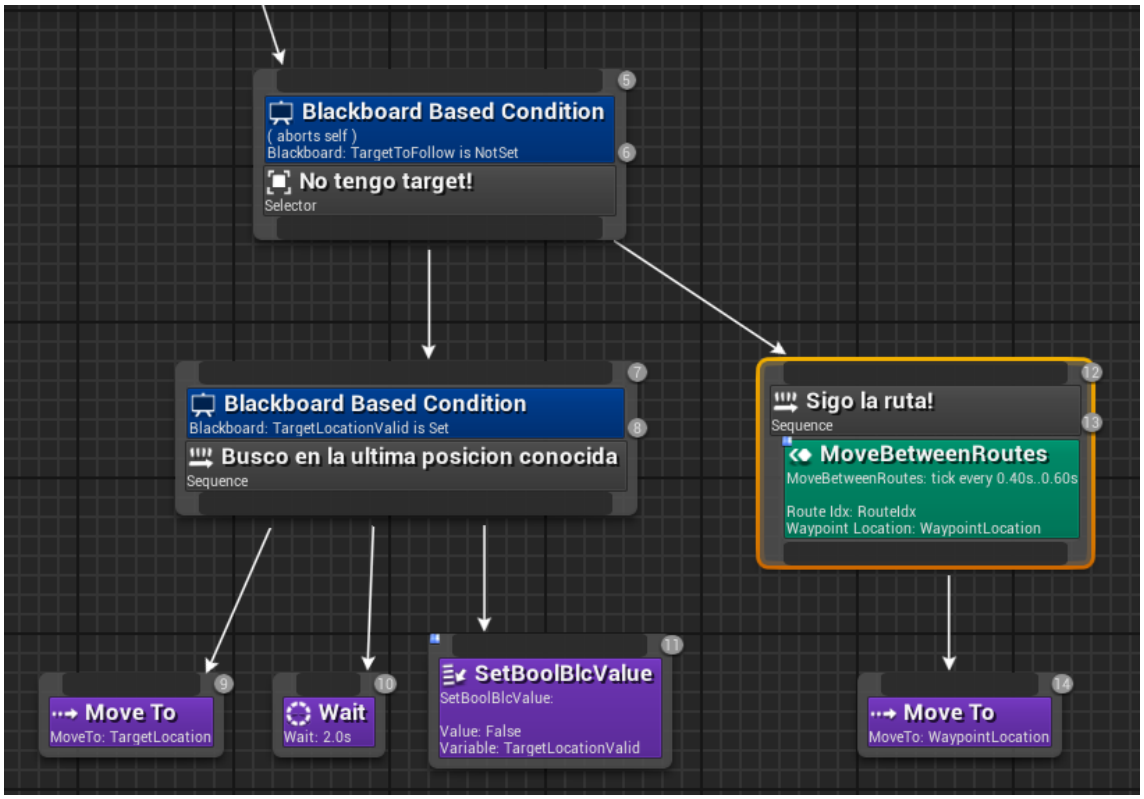
Si no está en distancia escribimos NULL en **TargetToFollow** y si no hacemos lo que hacíamos anteriormente y escribimos el actor en **TargetToFollow** y su posición en **TargetLocation**.

Con nuestro servicio modificado podemos deshacernos del decorador **CheckTargetDistance**.

Ahora en el lado derecho del árbol tenemos que hacer que siga la ruta configurada y además que si acaba de perder al target vaya a la **TargetLocation** y espere allí 2.5 segundos antes de seguir la ruta de nuevo.

Primero vamos a renombrar **HomeLocation** a **WaypointLocation** tanto en la blackboard como en la inicialización del **MyAIController** porque ahora ya no va a ser la posición de la casa sino el siguiente punto a seguir en la ruta, así cada cosa se llamará como lo que es, para no confundirnos.

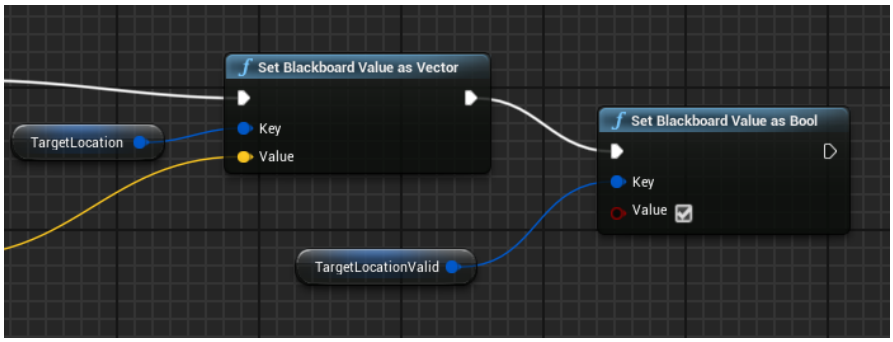
El subárbol derecho debería ser algo así:



Un **Selector – No tengo target!** Que va a intentar primero ejecutar la rama de la **Sequence – Busco la última posición conocida** y la **Sequence – Sigo la ruta!**

Como se puede observar hay que crear el **Service – MoveBetweenRoutes** que nos ira seleccionando que **Waypoint** es el siguiente, lo vemos un poco más adelante, y hay que añadir una variable de control a la blackboard **TargetLocationValid** que nos va a servir para saber cuándo hay que ir a buscar al target a la última posición conocida.

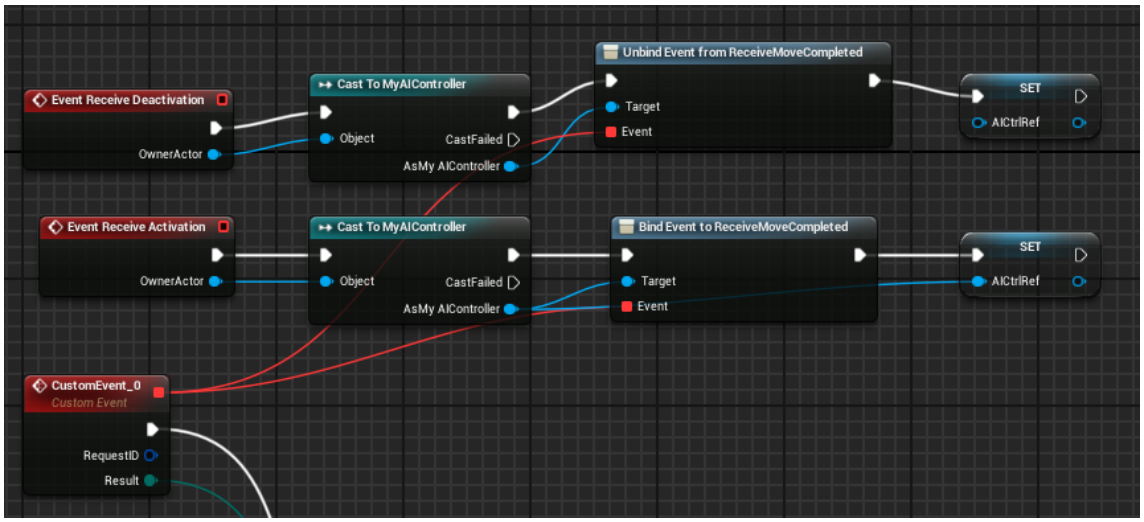
Añadimos la variable **TargetLocationValid** tipo bool a la blackboard, al ser un boolean si no le asignamos un valor este será false al pedirlo, y lo que hacemos es que cada vez que se asigne **TargetLocation** en el **Service – TargetSearch** ponemos esta variable a true para saber que a esa posición no hemos ido.



¿Qué vamos a hacer en el **Service – MoveBetweenRoutes?** Vamos a utilizar parte de lo aprendido en la Practica 1 pero dándole una vuelta más. Básicamente lo que ha de hacer este servicio es escribir en **WaypointLocation** la posición del siguiente **Waypoint** de la ruta y cuando lleguemos a él seleccionamos el siguiente.

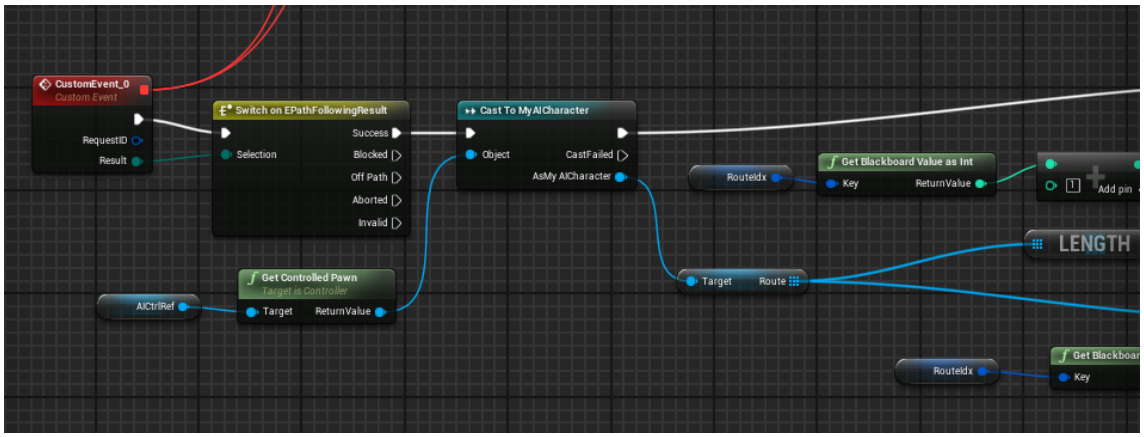
Lo primero que necesitamos es una variable nueva en la blackboard para almacenar el nodo actual de la ruta **RouteIdx** del tipo int. Una vez lo tengamos creamos 3 variables en el servicio un **AICtrRef** con hacemos siempre y una variable **Blackboard Key Selector** para cada dato de la blackboard que necesitamos en este caso **RouteIdx** y **WaypoitLocation**, ponemos estas dos últimas como Editables para configurarlas en el BT.

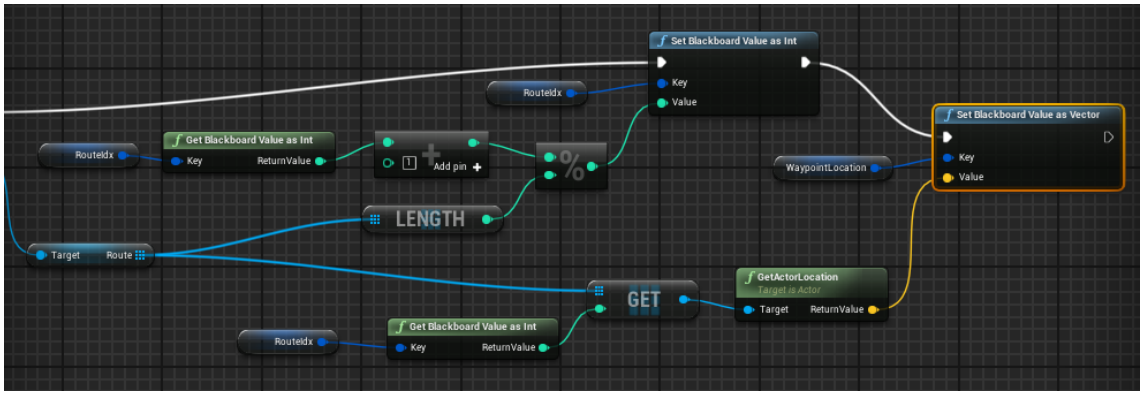
Lo que vamos a hacer es utilizar el evento que ya conocemos **ReceiveMoveCompleted** para recibir cuando se completa un movimiento, en nuestro el caso el movimiento que hagamos al **WaypointLocation**, en el **Event Receive Activation** nos apuntamos al evento y en el **Event Receive Deactivation** nos desapuntamos para no recibir otros move completed que no nos interesen:



La idea es que cada vez que recibamos una notificación de un **Move To** completado con **Success** vamos a calcular el siguiente Waypoint.

Con el nodo **Switch On EPathFollowingResult** vamos a poder saber el tipo de resultado recibido en caso de ser **Success** sacamos la **Route** del **MyAICharacter** para saber el número de Waypoints de la ruta y poder hacer **(RouteIdx + 1) % Length(Route)** y así calcular el nuevo índice de Waypoint al que ir. Escribimos en la blackboard tanto **RouteIdx** como el **WaypointLocation** y esperamos al siguiente evento **ReceiveMoveCompleted**.





Ya tenemos todo lo necesario, configuramos el BT y probamos a ver como se mueve ☺

¿Qué más podríamos hacer?

- Modificar la velocidad del Character dinámicamente para que cuando está haciendo la ruta vaya andando y cuando persigue al jugador vaya corriendo.
- Añadir lógica para no seguir la ruta por donde estaba antes de ver al jugador y elegir el siguiente punto con algo más de sentido, esto se podría usar utilizando el **Enviroment Query System** que veremos en futuras prácticas.
- Crear Smart Objects con los que puedan interactuar las IAs para crear distracciones que pueda aprovechar el jugador.

PROBLEMAS VERSIÓN >= 4.18

- Al crear una Task con C++ el proyecto da un error de linkado.

  Para solucionarlo añadir la siguiente línea en el fichero 'ProjectName'.Build.cs

  PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore", "AIModule", "GameplayTasks" });

- La función TickTask de la Task no se me invocaba hasta que he añadido este constructor para establecer la variable miembro

  ```
  UMoveToEnemyBTTaskNode::UMoveToEnemyBTTaskNode() {
  bNotifyTick = true;
  }
  ```

PROBLEMAS VERSIÓN >= 4.23

- En la “Parte 4: Ampliando el árbol en C++: Chase”, cuando creamos la función ChechNearbyEnemy, hacemos un testeo de esfera contra Pawns para ver cuál de ellos es el Player; en versiones anteriores se podía obtener el primer player del mundo con la llamada:

  ACharacter* pCharacter = UGameplayStatics::GetPlayerCharacter(GetWorld(), 0);

  Podíamos utilizar este Character que nos devolvía para compararlo con los actores con los que había colisionado nuestra esfera pero ya no, debemos comparar contra un Pawn. Ahora hay que usar:

  APawn* pPlayerPawn = UGameplayStatics::GetPlayerPawn(GetWorld(), 0);