Ejercicio Introducción comportamientos

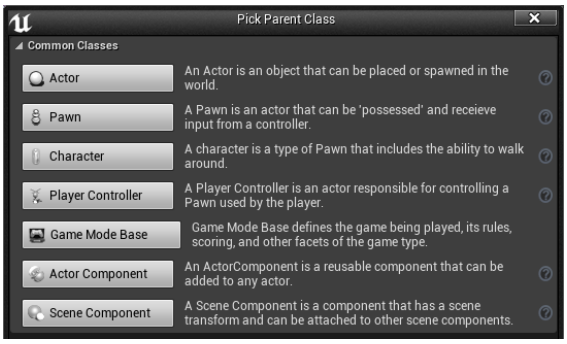
Parte 1: Movimiento aleatorio

El objetivo de la primera parte de esta práctica es familiarizarse con técnicas básicas de IA y darnos cuenta de lo que vamos necesitando para que la IA parezca cada vez más inteligente. Nuestro objetivo va a ser por tanto empezar por tener algo tan sencillo como un personaje controlado por la IA moviéndose de forma aleatoria indefinidamente.

Como base vamos a usar el **template "third person"**, creamos un nuevo proyecto y usamos esta template, como de momento no vamos a tocar nada de código nativo podemos usar la **template de blueprint**. La inteligencia artificial se lleva muy bien con los blueprint y es una buena idea que este en datos fácilmente editables, además con la potencia que ofrece UE4 no tendremos que implementar ninguno de los sistemas base que se usan actualmente en la industria para modelar una IA, trataremos de entender porque se usan y cómo funcionan.

En esta template se nos crea automáticamente un **ThirdPersonCharacter** que tiene el input ya configurado para poder movernos, saltar, etc. Y una cámara en tercera persona. Lo vamos a dejar tal como está para poder seguir a las IAs que creemos y movernos por el mapa cómodamente.

Ahora tenemos que crear un Character para que nuestra IA lo maneje, para ello en la carpeta de los blueprint del proyecto, en el panel **Content Browser**, hacemos clic con el botón derecho y se abrirá un menú desplegable con opciones para crear tipos de blueprints.



Seleccionamos **Character** y esto nos creara un nuevo Character en nuestra carpeta de blueprints al que llamaremos **"AI_Character"**.

Para configurar el **AI_Character** hacemos doble clic sobre él y se abrirá el editor:

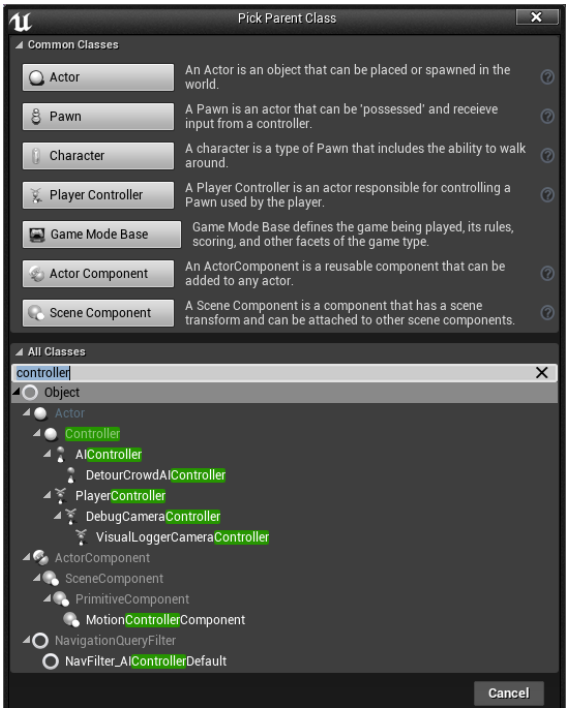
1. Seleccionamos el componente **Mesh** en el panel de **Componentes**.
2. En el panel **Details** buscamos la categoría **Mesh** y asignamos el **SK_Mannequin** al **Skeletal Mesh**. De esta forma nuestro Character usara esta representación visual.

3. Movemos la **Mesh** hacia abajo para que este centrada en la capsula y la **rotamos** de forma que mire hacia donde apunta la flecha azul.
4. Ahora tenemos que configurarle las animaciones, buscamos en el panel **Details** la sección **Animation** y le asignamos el **ThirdPerson_AnimBP** a la propiedad **Animation Blueprint Generated Class**.
5. Seleccionar el **CapsuleComponent** del panel **Components** y ajustar en el panel **Details**, la **Capsule Half Height** y el **Capsule Radius** para que se ajusten al **Skeletal Mesh**.

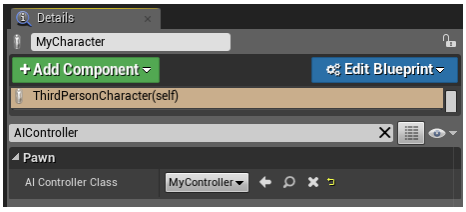
6. Por último, en **Components** seleccionamos el componente **Character Movement** y en el panel **Detail** configuramos el **Nav Agent Radius = 42** y el **Nav Agent Height = 192** (lo que mide la capsula de alto).

Una vez tenemos nuestro **AI_Character** lo arrastramos al mapa para crear una instancia y la llamamos “MyCharacter”. Además, vamos a necesitar un **AIController**, en UE4 existe una jerarquía de clases que Actor/Controller que permiten controlar un Pawn, así es como el player puede controlar cualquier par mediante su PlayerController, en el caso de la IA tenemos el AIController.

El **AIController** nos ofrece funcionalidad para mover al Pawn y usar un Behavior Tree para controlar las decisiones (pero eso lo veremos más adelante). Creamos el AIController y lo llamamos “MyController”. Hacemos lo mismo que para el Character, botón derecho en la carpeta de blueprints, pero en este caso no vamos a crear una **Common Class,** sino que vamos a tener que buscar en **All Classes** el AIController:



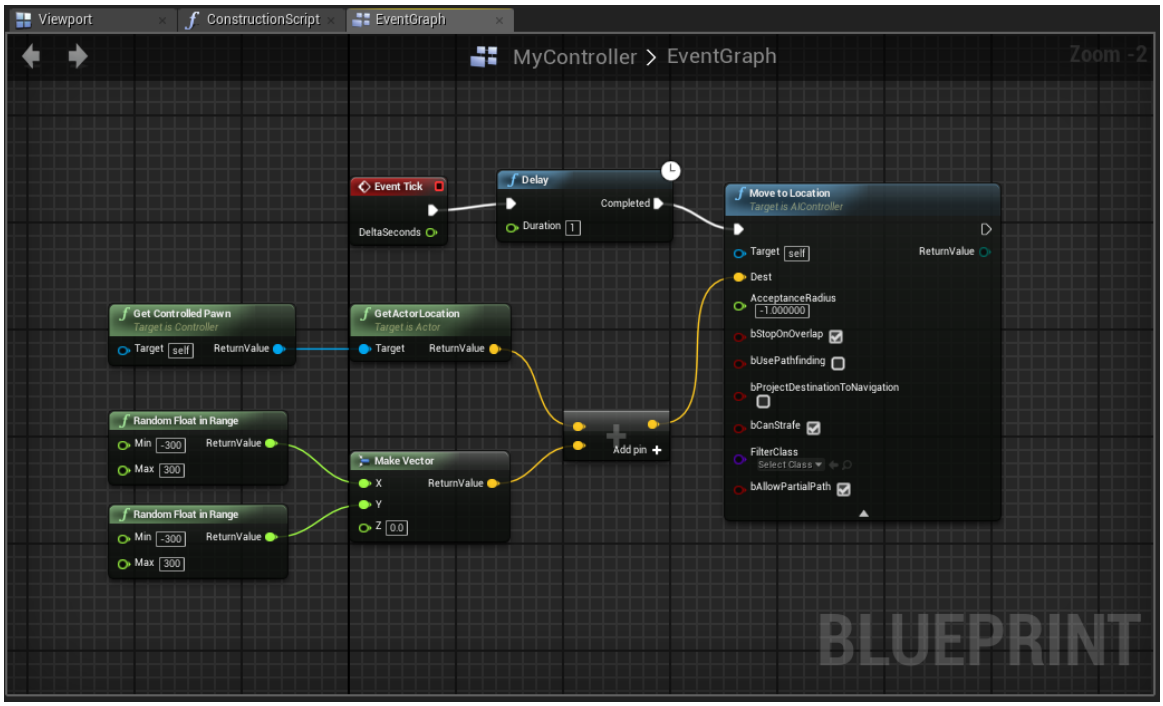
Una vez tenemos nuestro AIController tenemos que asignárselo a su Character, para ellos seleccionamos la entidad “MyCharacter” y ponemos en el buscador de propiedades (panel **Details**) AIController y le asignamos “MyController”. Con esto lo que conseguimos es que cuando al Pawn no se le haya hecho Possess de ningún controlador se le haga con una instancia de “MyController”.



Ya tenemos el proyecto configurado para poder manejar “MyCharacter” con “MyController” así que abrimos “MyController” en el **Content Browser** y abrimos el **EventGraph**:

El objetivo es mover el Pawn de forma aleatoria. Lo vamos a hacer de la manera más sencilla para tener un ejemplo muy básico de IA.

1. Creamos un **Event Tick** si no está ya creado; este evento se llamará todos los frames.
2. Conectamos un **Delay** al Event Tick y lo configuramos a **1 segundo**. Queremos que se mueva a una posición diferente cada segundo para hacer el movimiento aleatorio.
3. Después del Delay añadimos un nodo **Move To Location**, será el que le diga al AIController a donde mover el Pawn. Como no estamos usando pathfinding (por el momento) desactivamos el uso de pathfinding.
4. Para decidir dónde va el Pawn cada segundo de forma aleatoria vamos a usar **Get Controlled Pawn** para obtener el Pawn al que controla el AIController y conseguir la posición actual del Pawn y hacer un movimiento aleatorio usando como base esta posición.
5. Calculamos un vector aleatorio en X e Y dejando la Z a 0 para no modificar la altura de la posición destino. Generamos dos números aleatorios con **Random Float In Range** con el rango **[-300,300]** y usamos un **MakeVector** para componer el offset a aplicar a la posición del Pawn. Sumamos la posición más el Offset con un **vector+vector**. Conectamos esta posición calculada a la entrada **Dest** del nodo Move To Location y así tenemos el destino aleatorio que queríamos.



Ya solo queda dar al play y ver como el Pawn se mueve ☺. Esto nos sirve como ejemplo muy básico de como poder mover un Pawn con un AIController, en el caso de que la IA que necesitáramos fuera esta (lo dudo) pues sería tan sencillo como esto. Desgraciadamente va a tener que complicarse un poco.

Antes de pasar a la siguiente parte de la práctica vamos a pararnos un poco en el nodo **Move To Location**, este es un nodo que se va a usar mucho si nuestra IA se va a mover. Aparte de los parámetros que hemos configurado anteriormente (desactivar el pathfinding) nos ofrece más opciones:

- *AcceptanceRadius*: Configura el radio en el que se considera que el movimiento ha llegado al destino, se puede usar por ejemplo para hacer un chase a otro Pawn y quedarnos a X distancia de él.
- *bStopOnOverlap*: Hace que el movimiento termine cuando el radio del Pawn contiene el punto destino, hace que no se tenga que llegar exactamente al punto.
- *bUsePathFinding*: Configura si el movimiento va a usar la NavMesh (lo veremos más adelante).
- *bProjectDestinationToNavigation*: Lo que hace es pasar la posición de destino al sistema de navegación antes de empezar el movimiento para saber si la posición está dentro o no de una zona jugable.
- *bCanStrafe*: Determina si la IA puede atravesar diagonalmente la NavMesh.
- *FilterClass*: Nos permite asignar al movimiento instancias del tipo AreaClass que se usan para restringir el movimiento de los Pawns, zonas exclusivas o prohibidas.

Resumiendo, hemos aprendido a configurar un proyecto con un Pawn controlado por un AIController y como mandar instrucciones de movimiento al AIController desde el Event Graph.

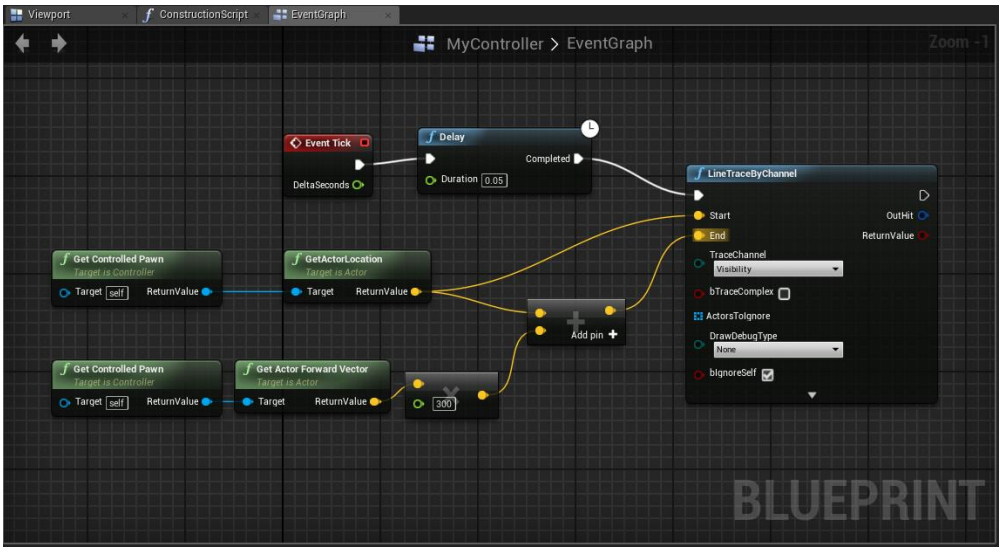
Parte 2: Esquiva de obstáculos básica

En la primera de la práctica hemos creado una IA que mueve un Pawn de forma aleatoria sin tener en cuenta el entorno, pero ¿y si quisiéramos que nuestro Pawn si tuviera en cuenta el entorno? Pues utilizaremos el Ray Tracing.

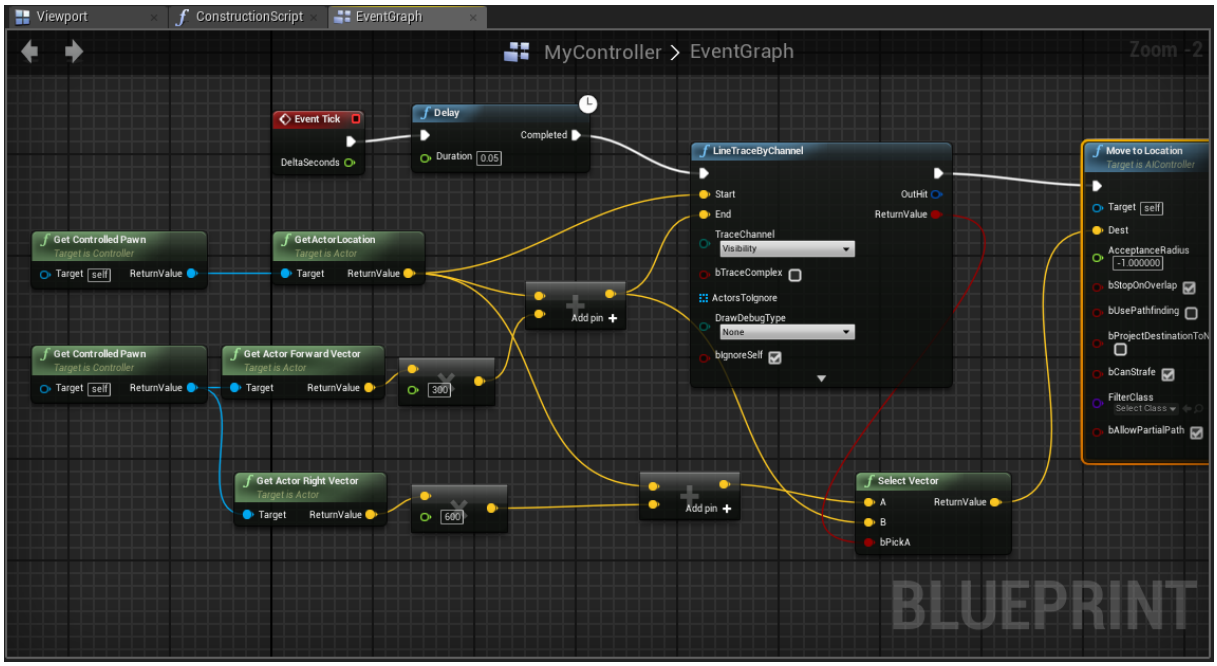
El Ray Tracing se usa para lanzar rayos (pueden ser también esferas, capsulas, etc.) para saber cómo es el entorno que nos rodea. Podemos sacar información como la normal de la cosa con la que colisionamos, su entidad, etc.

El objetivo es mover al Pawn hacia delante hasta que se encuentre con un obstáculo, entonces giraremos a la derecha.

1. Borramos todos los nodos que usábamos para calcula la posición de destino y solo dejamos el **Delay** y el **Move To Location**.
2. Bajamos el tiempo del **Delay** a **0.05 segundos** para que sea más reactivo.
3. Creamos un nodo **Line Trace By Channel** que es el encargado de lanzar un rayo y decirnos si ha colisionado con algo.
4. Volvemos a pedir la posición del Pawn con **Get Controlled Pawn** y **Get Actor Location** y se lo metemos a **Line Trace By Channel** como **Start**.
5. Queremos lanzar un rayo hacia delante por lo que necesitamos calcular el vector dirección y aplicarle un multiplicador, para ello usamos **Get Controlled Pawn** y **Get Actor Forward Vector** y lo **multiplicamos por 300** por ejemplo.
6. Sumamos el vector dirección calculado a la posición del Pawn y tenemos el **Dest** de **Line Trace By Channel**.



7. El nodo **Line Trace By Channel** devolverá en **ReturnValue** si ha habido colisión o no, en caso de que no haya colisión queremos movernos hacia delante y en caso de colisión hacia la derecha.
    a. Primero tenemos que calcular el "vector derecha" con el nodo **Get Actor Right Vector** y lo multiplicamos por **300**.
    b. Después conectamos un nodo **Select Vector** a la salida **ReturnValue** de **Line Trace By Channel** y en **A** conectamos el vector derecha + posición y en **B** el vector de frente + posición.
    c. La salida del **Select Vector** se la metemos al **Move To Location** como destino.



Ya solo queda pulsar play y observamos como la IA avanza hasta colisionar con una pared y gira a la derecha. En el mapa actual se quedará dando vueltas alrededor de la pared exterior esquivando la zona centra en caso de que se la encuentre al principio.

Resumiendo, hemos aprendido como tirar rayos para analizar el entorno y modificar el movimiento para esquivar obstáculos. Hay que tener en cuenta que los rayos son bastante costosos y dependiendo de la maquina destino nos podremos permitir más o menos por frame.

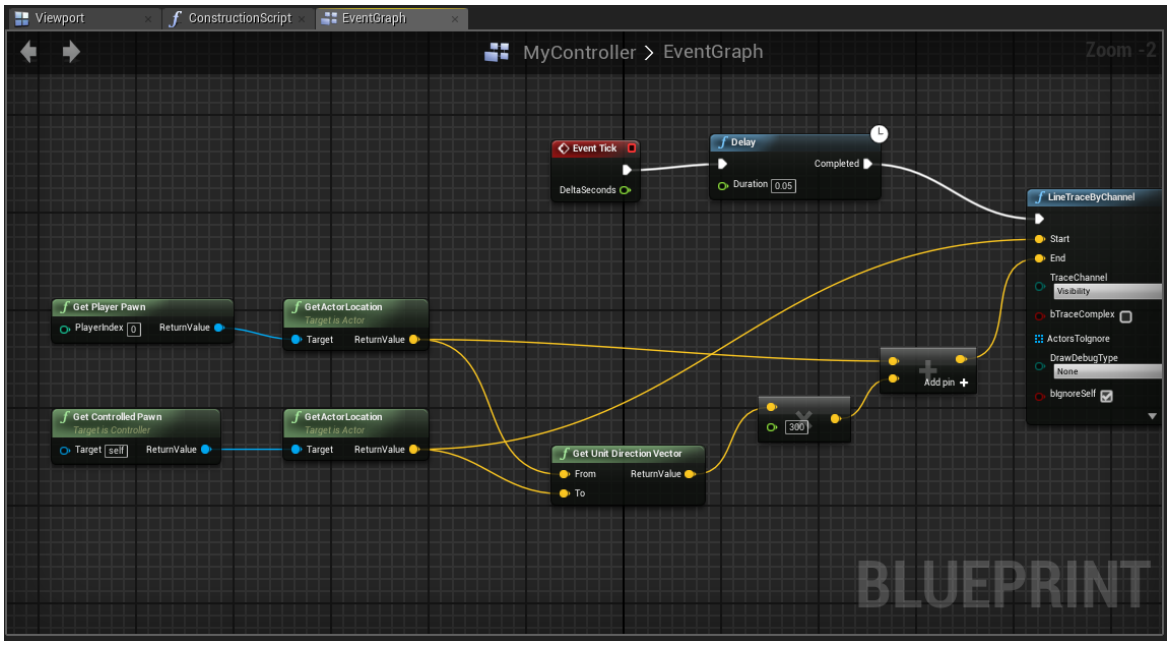
Parte 3: Seguir al jugador (chase)

Ahora lo que vamos a hacer es seguir al jugador a una cierta distancia, esquivando obstáculos en el camino.

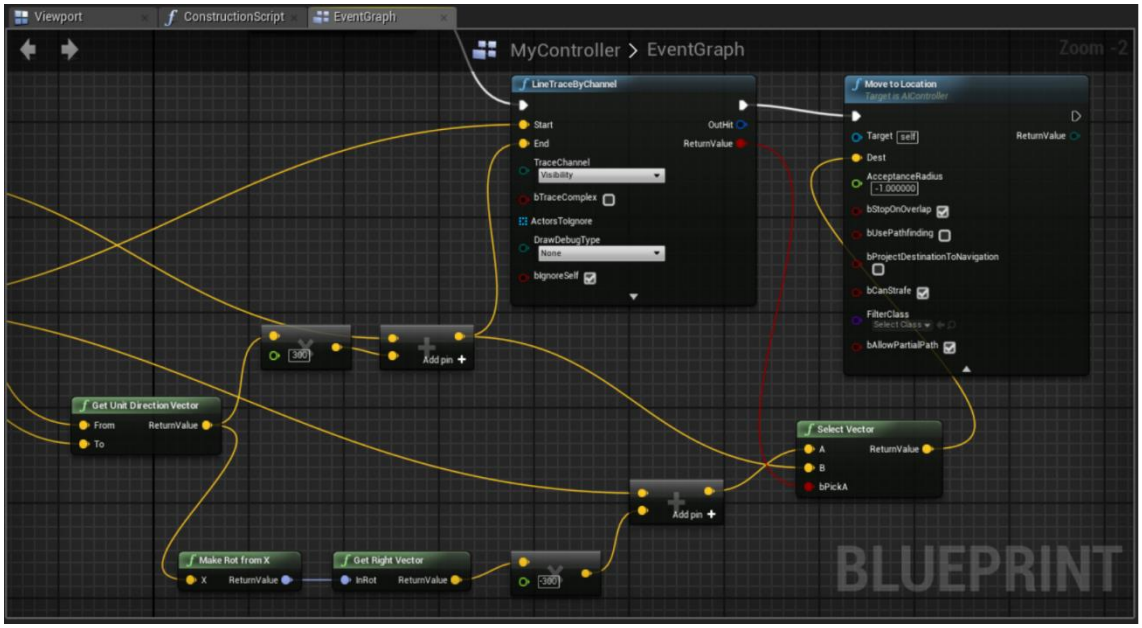
Vamos a ver como utilizando solo **Move To Location** y **Line Trace By Channel** podemos hacer muchas cosas distintas, en UE4 habría mejores formas de implementar estos comportamientos, pero lo usamos como base para luego complicar las cosas.

El objetivo es mover al Pawn hacia el jugador hasta que se encuentre con un obstáculo, entonces giraremos a la derecha.

1. Primero tenemos que calcular las entradas de **Line Trace By Channel**, el origen será la posición del Pawn (usamos **Get Controlled Pawn** y **Get Actor Location** para conseguir).
2. El destino de **Line Trace By Channel** será una posición a 3 metros del player en dirección a nuestro Pawn
    a. Con **Get Player Pawn** obtenemos el Pawn del player y le pedimos su posición con **Get Actor Location**.
    b. Para calcular la dirección del player a nuestro Pawn usamos **Get Unit Direction** Vector (From: Player Pawn To: Nuestro Pawn). Una vez tenemos la dirección normalizada la multiplicamos por 300 y se la sumamos a la posición del Pawn del player.



3. Ahora tenemos que calcular que destino pasar al **Move To Location**, para ello vamos a utilizar la información que nos devuelve el **Line Trace By Channel** de forma que si no hay colisión vamos hacia el punto deseado y si hay colisión debemos ir hacia la derecha como en el ejemplo anterior.
    a. Como ya tenemos calculada la dirección normalizada del player a nuestro pawn vamos a sacar la rotación de ese vector con **Make Rot From X** con este nodo generamos un rotator a partir de un vector frontal.
    b. De esta rotación sacamos fácilmente el vector derecha con **Get Right Vector**.
    c. Lo multiplicamos por -300 y se lo sumamos a la posición de nuestro Pawn, de esta forma ya tenemos calculada la posición a la que movernos en caso de encontrar colisión.
4. Usamos el nodo **Select Vector** para seleccionar a donde ir en base al resultado de **Line Trace By Channel**, conectamos **ReturnValue** a **bPickA**, entonces conectamos en **A** la posición de escape hacia la derecha y en **B** la posición deseada para hacer el chase.



Pues ya tenemos todo listo, lanzamos el juego y vemos como la IA nos seguirá y se parar a 3 metros. Gracias al ray tracing ira esquivando los obstáculos que se encuentre.

Resumiendo, hemos aprendido como con muy poco esfuerzo con las herramientas que nos ofrece UE4 podemos hacer comportamientos de IA bastante interesantes. Un comportamiento chase es muy usado en los juegos, por ejemplo, en juegos tipo hack and slash donde los enemigos rodean y siguen al jugador y se van coordinando para lanzar ataques.