Ejercicio Pathfinding

Parte 1: Rutas aleatorias sobre waypoints

Imaginemos que lo que queremos ahora es tener unos puntos en el mapa que vigilar y queremos que nuestra IA seleccione uno de ellos de forma aleatoria vaya hasta allí y al llegar vuelva a elegir otro, de forma que de vueltas indefinidamente.

Lo que hemos visto hasta ahora está muy bien, pero eso de moverse tirando rayos no vale para mucho en este caso ya que lo que necesitamos es calcular un camino de la posición actual a un punto cualquiera, al que no sabemos si vamos a poder llegar solo girando a la derecha al encontrarnos una pared.

Necesitamos tener una representación del mundo más sencilla que la capa física para poder calcular caminos de un punto a otro, hay varias formas de representar el espacio (grid, waypoints, navmesh, etc.) y depende mucho del tipo de juego y del tiempo que tengamos para implementarlo (en el caso de no usar UE4). UE4 utiliza NavMesh que es prácticamente el estándar en la industria en representación espacial, en el pasado se utilizaba mucho más waypoints porque son más fáciles de implementar y ocupan menos espacio. Pero gracias a mucha investigación en generación de NavMesh a lo largo de los años se conocen técnicas bastante sencillas para generarlas por lo que cualquiera puede hacer un algoritmo de generación de cero.

La NavMesh nos aporta más información que los waypoints porque tiene información de los bordes de la zona navegable; con los waypoints solo sabíamos si podíamos ir de uno a otro, pero no sabíamos si todo el interior de 3 waypoints conectados era navegable, por lo que durante la navegación habría que hacer ray tracing, tal como hemos hecho en la parte 2 para intentar optimizar el camino de manera que no fuera un movimiento tan robótico.

Primero vamos a hacer pruebas con el generador de NavMesh de UE4 para familiarizarnos con él y después montaremos el comportamiento de ir a puntos seleccionados de una lista de puntos metidos a mano en el editor.

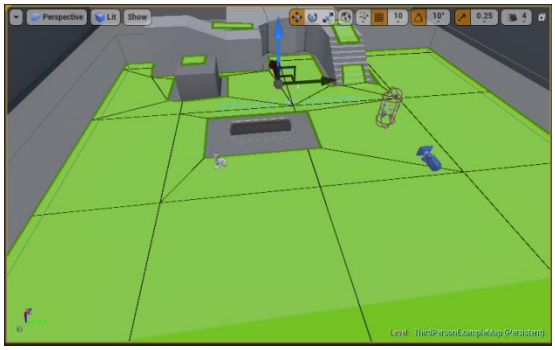
Para que se genere una NavMesh en el nivel actual tenemos que crear un **NavMeshBoundsVolume** que abarque toda la zona donde queremos que la IA pueda navegar. Este tipo de volumen se encuentra en **Modes/Volumes** y simplemente tenemos que arrastrarlo a nuestra escena. Vamos a crear un nuevo proyecto con el template “third person” y creamos el **NavMeshBoundsVolume**.

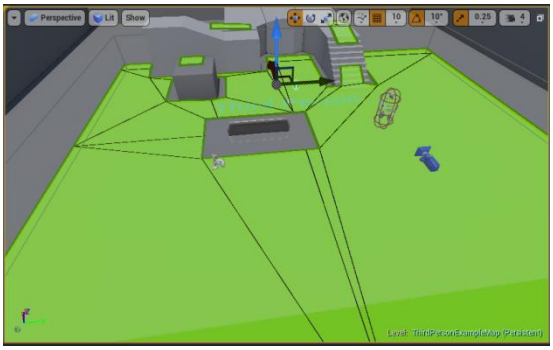
Para mostrar la información de navegación podemos marcar la opción **Navigation** o usar el atajo de teclado **P**. Si queremos verla ingame sacamos la consola y ponemos ***‘show navigation’***.

Ejercicio pathfinding

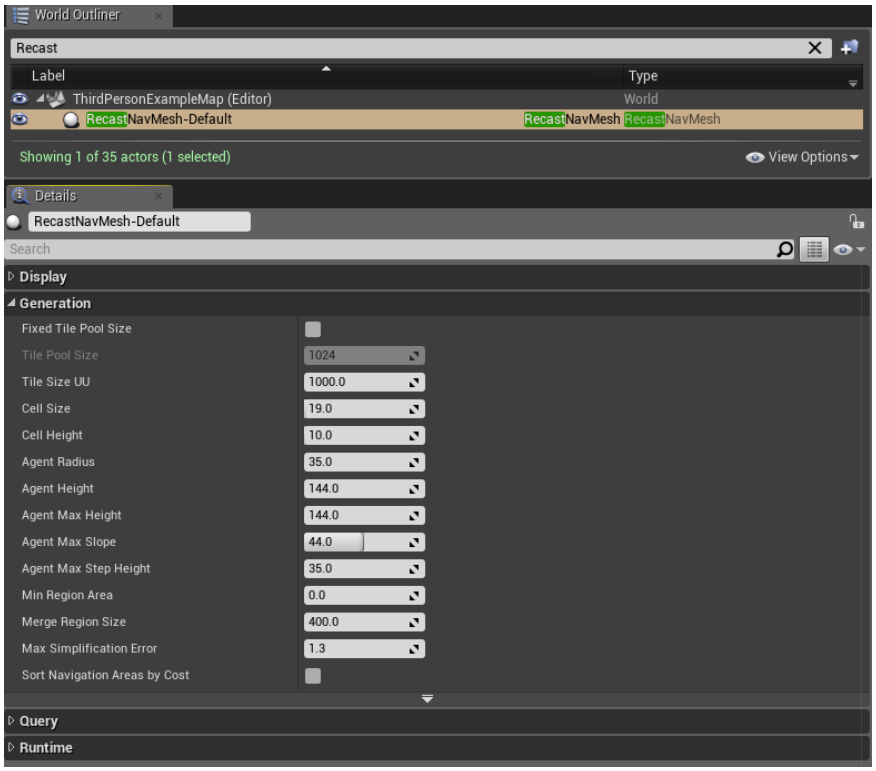
La configuración de la generación de la NavMesh está en un objeto que se llama **RecastNavMesh-Default** en el **World Outliner**. En él se pueden configurar los valores por defecto para la generación de la NavMesh en este nivel, los parámetros más importantes son:

- *Cell Size*: Es el tamaño del voxel (un pixel en 3d, un bloque de minecraft ☺) que representa un espacio navegable. Este tamaño se refiere al tamaño del suelo del voxel es decir ancho y profundidad. Esto lo que controla es la resolución de la NavMesh, cuanto más pequeño sea más fiel a la colisión real será la NavMesh, pero más costosa de generar. Al ser la generación offline esto nos da un poco igual a no ser que queramos regenerar la NavMesh en caliente por destrucción de objetos dinámicos, por ejemplo.
- *Cell Height*: Altura del voxel, este valor se debe corresponder con la máxima altura que puede subir una entidad andando, esto quiere decir que todo escalón más bajo que esta altura no existe para la NavMesh.
- *Tile Size UU*: Tamaño de celda. Si bajamos este valor el tiempo de generación aumenta, pero la NavMesh generada es más limpia, a la izquierda vemos una NavMesh generada con 560 y a la derecha con 1500.

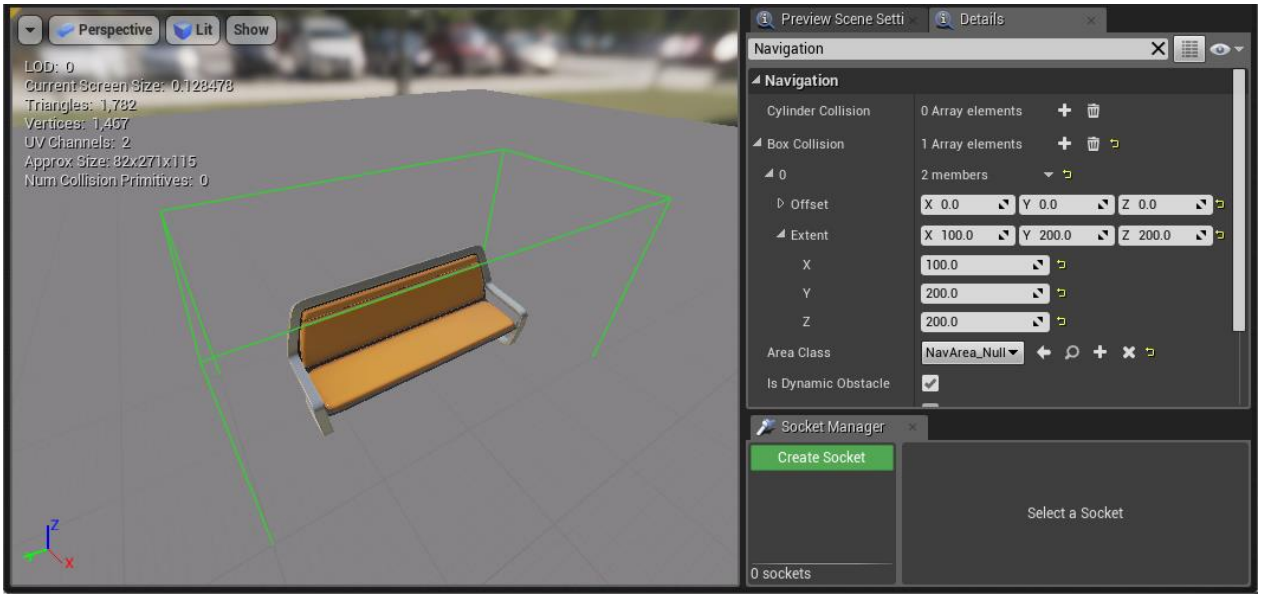




- *Min Region Area*: Quita de la NavMesh las regiones que están aisladas que sean menores que el tamaño configurado.

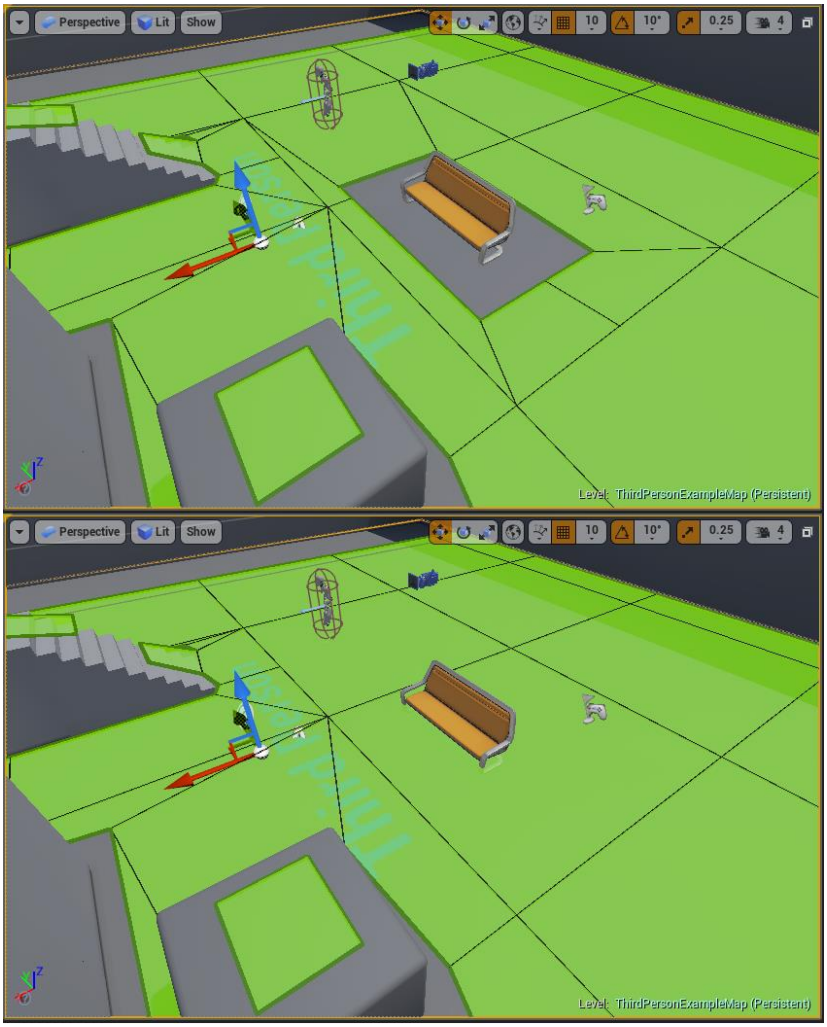


Si hay algún obstáculo en la NavMesh, por ejemplo, un banco nos va a interesar que la NavMesh sepa que es zona no es navegable para no tener que estar haciendo chequeos ingame. Si editamos un prop en la sección Navigation podemos configurar cilindros o cajas con información de navegación; para hacer una zona no navegable le asignamos el **Area Class** del tipo **NavArea_Null** (para poder elegir el **AreaClass** como **NavArea_Null** tenemos que seleccionar previamente el *CheckBox* **Is Dynamic Obstacle**). Veremos más adelante que es esto de las Area Class.



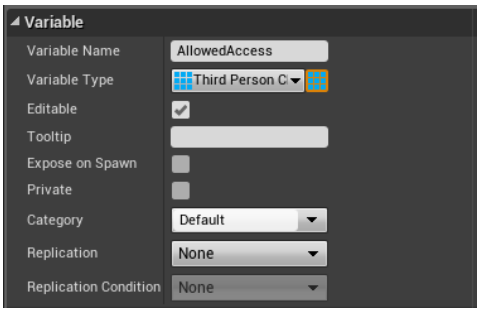
Ejercicio pathfinding

En la siguiente imagen podemos ver como el banco genera una zona no navegable en la NavMesh:

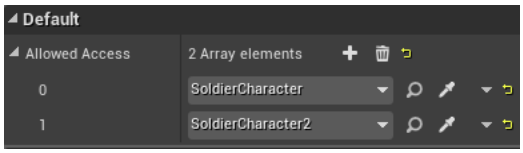


El primer objetivo va a ser crear waypoints que se puedan poner en el nivel para decirle a las IAs los puntos a los que ir. Además, daremos soporte para poder configurar que IAs pueden usar cada waypoint.

1. En la carpeta de **Blueprint** con el botón derecho creamos un nuevo blueprint del tipo **Target Point** (lo usamos como base por el sprite que usa como representación) y lo llamamos Waypoint.
2. Editamos el Waypoint y creamos una nueva variable array del tipo **AI_Character** para poder almacenar la lista de carácter que pueden usar el waypoint y la configuramos como **Editable** para poder acceder a ella desde fuera del objeto.



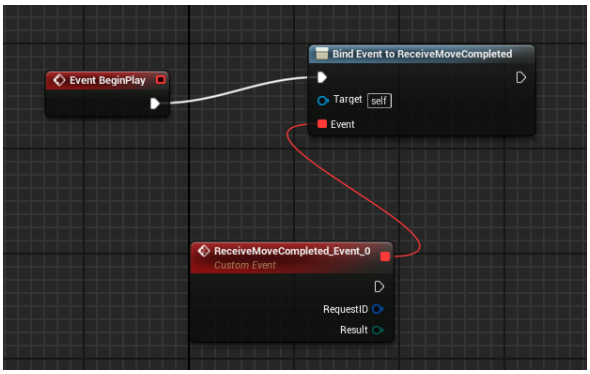
3. Ahora ponemos varios Waypoints en el nivel.
4. Ponemos dos **AI_Character** en el mapa “Soldier1” y “Soldier2”.
5. Vamos a configurar los Waypoints para que algunos no puedan ser usados por alguno de los dos personajes que hemos colocado en el mapa. Esto lo hacemos rellenando el array **AllowedAccess** que habíamos creado con los dos characters, o solo uno de ellos.



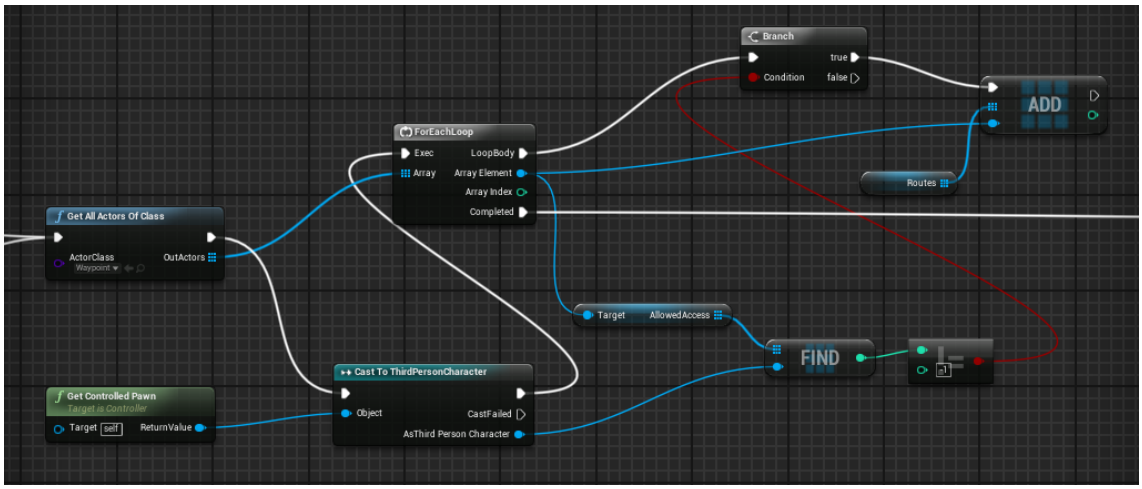
Ya tenemos creado nuestro Waypoint que nos permite configurar que characters lo pueden usar así que ahora vamos a crear el comportamiento que elija a que Waypoint ir.

El segundo objetivo va a ser crear un comportamiento en el AIController que elija un Waypoint aleatorio, vaya hasta él y al llegar elija el siguiente y así indefinidamente.

1. Creamos un **AIController**, lo llamamos “SoldierAI” y se lo asignamos a “Soldier1” y “Soldier2”.
2. Abrimos el **EventGraph** de “SoldierAI” para editar el comportamiento, lo primero es que no vamos a querer implementar el comportamiento en el **Event Tick** ya que lo que queremos es ir a un punto y cuando hayamos llegado ir a otro. Para esto tenemos que escuchar el evento **ReceiveMoveCompleted** que es un evento que lanza el componente de movimiento cuando se completa una orden de movimiento, por ejemplo, un **Move To Actor** que será el que utilicemos. Para registrarnos al evento:
    a. Registrarse al evento solo hay que hacerlo al empezar así que lo haremos en el evento **Begin Play** que se lanza cuando el controlador entra en juego.
    b. Para el registro utilizamos el nodo **Bind Event to** que se utiliza para registrarse a cualquier evento en este caso a **ReceiveMoveCompleted**.
    c. Conectamos un nodo tipo **Custom Event** a la entrada evento del nodo **Bind Event to ReceiveMoveCompleted** que será donde recibamos el evento.

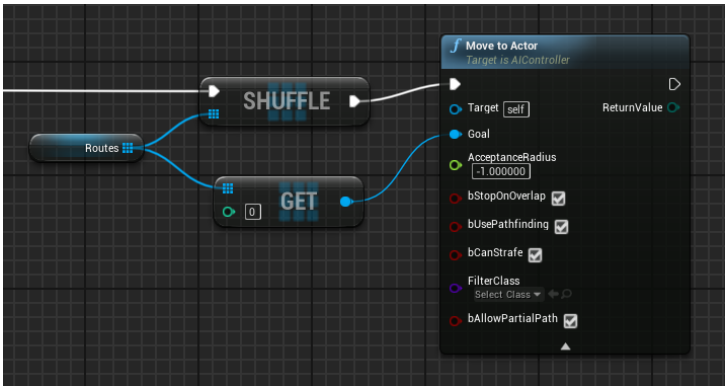


3. Ahora lo primero que hay que hacer es decidir a que waypoint vamos a ir. Cogemos todos los actores del tipo **Waypoint** con el nodo **Get All Actors Of Class** y le configuramos el tipo **Waypoint**. Este nodo nos devuelve un array con todos los actores del tipo seleccionado. Conectamos tanto la salida del **Custom Event** como la del Bind Event to al **Get All Actors Of Class** para que se elija un nuevo punto donde ir en el **Begin Play** y cada vez que se llegue a un punto.
4. De esa lista con todos los waypoints tenemos que extraer los que puede usar nuestro Pawn, pedimos el Pawn con **Get Controlled Pawn** y le hacemos un **Cast to AI_Character** para poder compararlo con los de la lista **AllowedAccess** de cada waypoint.
5. Iteramos cada elemento de la lista de waypoints y miramos si esta nuestro Pawn en su lista de **AllowedAccess**. Iteramos con un nodo **For Each Loop** sobre la salida de **Get All Actors Of Class** y cada loop (**Loop Body**) comprobamos si el Pawn está en la lista **AllowedAccess** con el nodo **Find**. Si **Find** devuelve -1 entonces el Pawn no está en la lista, por lo tanto, comparamos que sea distinto de -1 y si es así lo metemos en un array temporal donde vamos a almacenar todos los waypoints válidos. Para ello creamos un array de waypoint que se llame “Routes” por ejemplo.



6. En el **Completed** del nodo **For Each Loop** ya tendremos los waypoints validos en el array **Routes**, lo único que nos falta por hacer es elegir un random. Es tan sencillo como hacer un **SHUFFLE** del array y elegir el primero para pasárselo como destino al

nodo **Move To Actor**. En este caso dejaremos marcado **bUsePathFinding** para que use la NavMesh y vaya por el mejor camino hasta el destino.



Ya solo nos queda probar que todo funcione correctamente y veremos a dos Pawns moviéndose a cada uno de los waypoints de forma aleatoria.

¿Cómo podríamos hacer que al llegar a cada punto esperara 3 segundos hasta volver a ir a otro waypoint?

Sería tan sencillo como poner un nodo **Delay** de 3 segundos entre el **Custom Event** y el nodo **Get All Actors Of Class**. Podríamos aprovechar para poner una animación de buscar o de cansancio, por ejemplo.

¿Cómo podríamos hacer que los dos no fueran al mismo nodo a la vez?

Esto sería algo más complicado que lo anterior pero también seria sencillo, simplemente tendríamos que crear una variable booleana editable en el Waypoint, llamada por ejemplo Locked, de forma que a la hora de filtrar los waypoints validos solo cogeríamos los que tiene esta variable a false. Además, tendríamos que poner la variable a true entre el **SHUFFLE** y el **Move To Actor** y restaurarla a false después del **Custom Event**.

Parte 2: Barro y Agua (Modificadores de la NavMesh)

Ahora que ya tenemos dos enemigos haciendo rutas aleatorias entre waypoints y parecen bastante inteligente porque saben esquivar obstáculos el equipo de diseño ha pensado que va a poner zonas de agua y barro en el escenario porque es muy molón. Nosotros como queremos que nuestra IA no parezca tonta de nuevo pasando por el barro o por el agua si no es necesario nos damos cuenta de que necesitamos meter información en la NavMesh para que el cálculo de paths sea algo más inteligente que buscar el camino más corto.

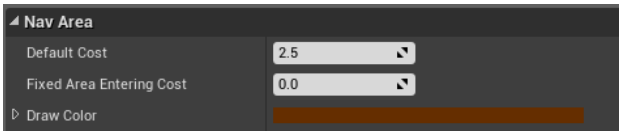
El objetivo es poder modelar de alguna forma el agua y el barro para que los cálculos de paths sobre la NavMesh los tengan en cuenta.

UE4 nos ofrece funcionalidad para realizar esto sin mucho esfuerzo, si lo hubiéramos tenido que hacer nosotros de cero a lo mejor no lo habríamos hecho, pero UE4 es lo que tiene.

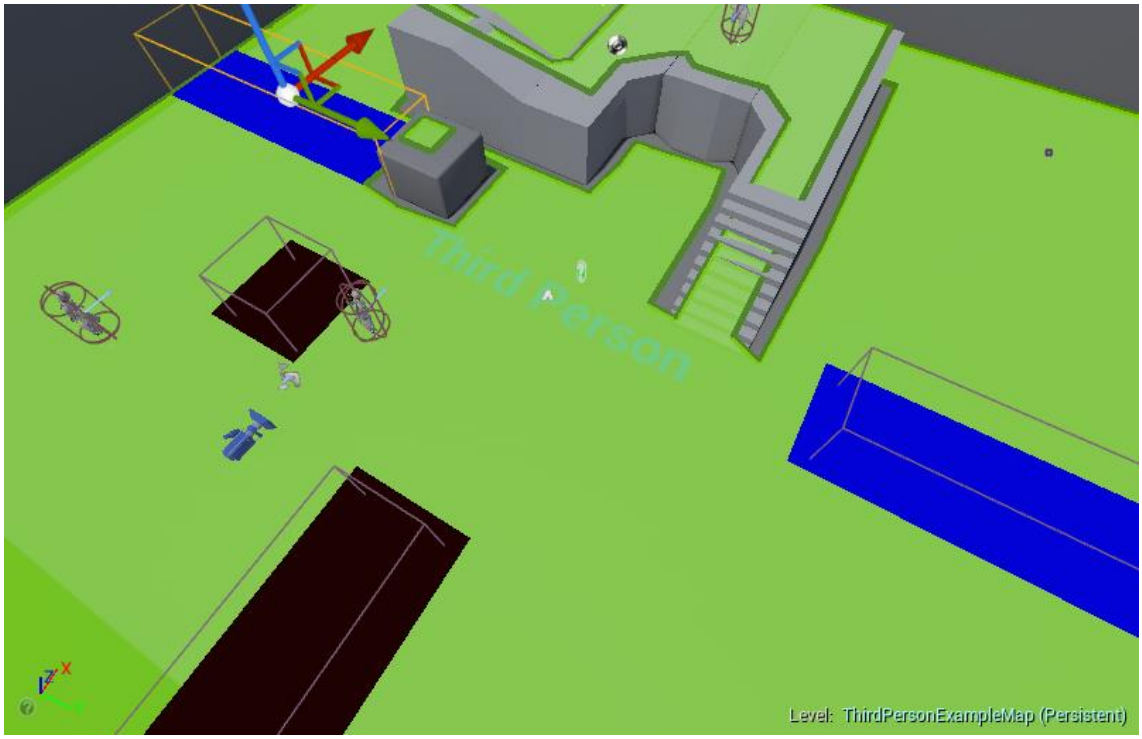
1. Buscamos **Nav Modifier Volume** en la ventana **Modes/Volumes**. Y Arrastramos unos cuantos a las zonas donde hay agua y barro. Hemos puesto 4 volúmenes de ejemplo.

2. Para configurar las propiedades de cada uno de los volúmenes tenemos que crear una **NavArea** por cada tipo que queramos, como queremos barro y agua creamos 2 y las llamamos “AreaOfMud” y “AreaOfWater”. Para crearla botón derecho en la carpeta **Blueprint** y elegimos **NavArea**.

3. Doble clic sobre **AreaOfMud** para configurarla:
    a. *Default Cost*: Multiplicador del coste por navegar por esa zona, se aplica al tramo de path que pase por la zona.
    b. *Fixed Area Entering Cost*: Coste fijo que se suma al entrar.
    c. *Draw Color*: Color de debug.



4. Configuramos el barro para que el **Default Cost** = 2.5 y el **Color** = marrón.
5. Configuramos el agua para que el **Default Cost** = 0.5, el **Fixed Area Entering Cost** = 500 (no nos gusta mojarnos) y el **Color** = azul.
6. Asignamos a cada **Nav Modifier Volume** el **NavArea** que queramos y debería quedar algo así:



Por un bug conocido en UE4 parece ser que los colores no se ven hasta que no salvas, cierras y vuelve a abrir el proyecto ☹

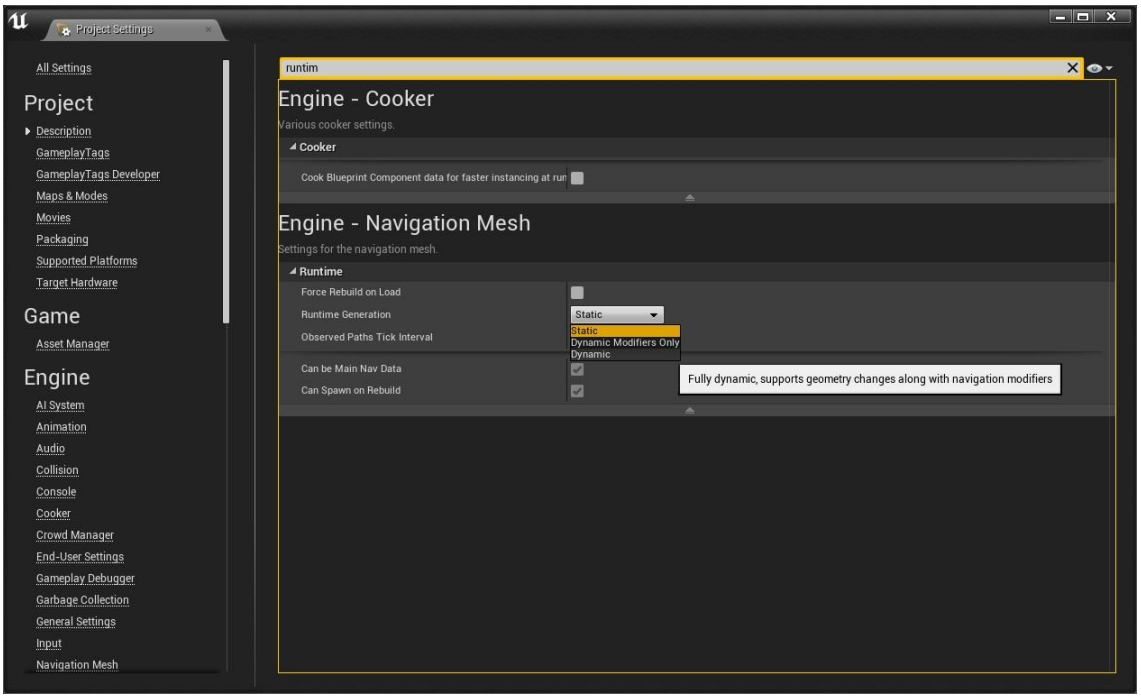
Si lanzamos la ejecución veremos como los Pawn esquivan cuando es posible los charcos y el barro, dependiendo de si el camino sin zonas con penalización no es mucho más largo. Todo esto se puede ajustar modificando los valores de penalización.

Ejercicio extra: NavMesh Dinámica

Probemos ahora a configurar nuestra NavMesh para que se recalcule automáticamente antes modificaciones del mapa.

Para ello debemos modificar en la configuración del proyecto, en el apartado Engine - Navigation Mesh el parámetro Runtime Generation a Dynamic.

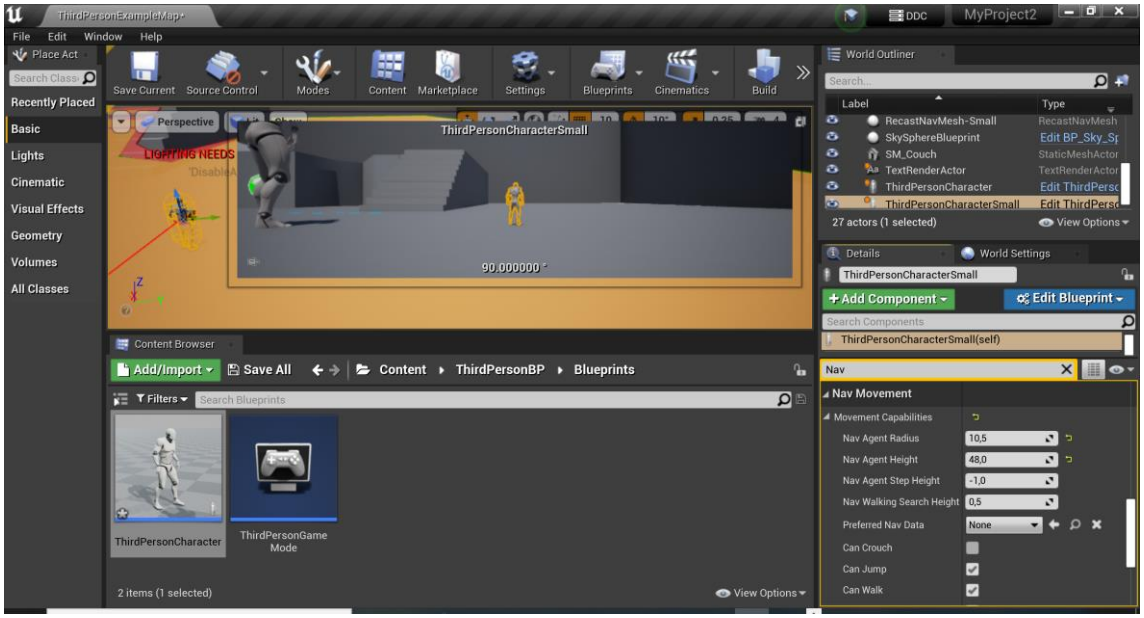
Podemos colocar algún objeto con física al que podamos mover (por ejemplo, disparándolo) y veremos cómo la NavMesh es recalculada automáticamente.



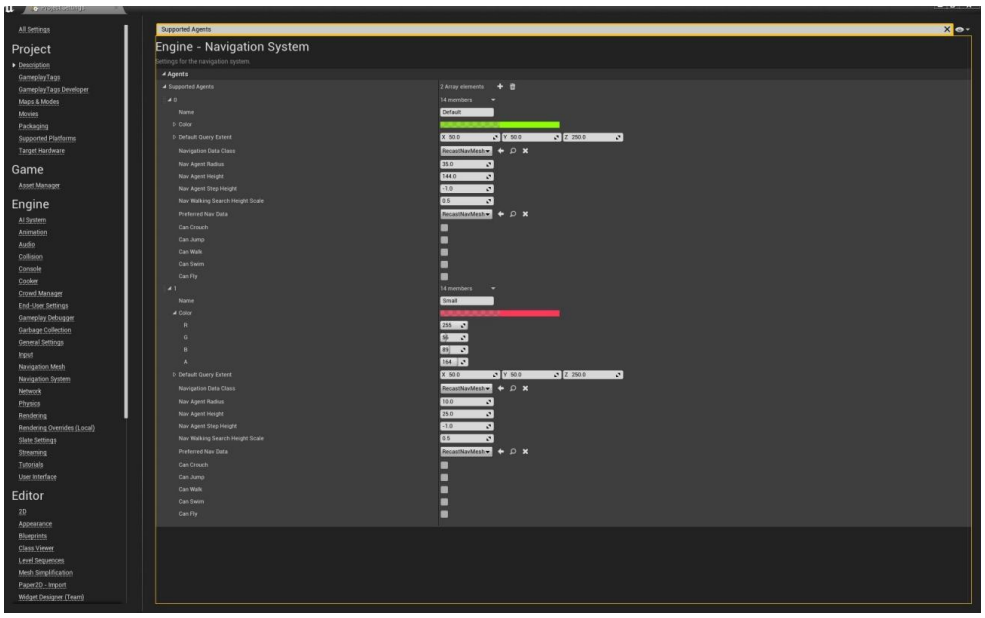
Ejercicio extra: Tipos de agente soportados por la NavMesh

Podemos hacer que para un mismo volumen de generación de NavMesh, se generen diferentes NavMeshes para diferentes tipos de agentes. Esto es útil cuando tenemos entidades en nuestro juego de diferente tamaño, con lo que querremos que las NavMeshes por las que naveguen se ajusten a la colisión en función del tamaño de cada una de ellas.

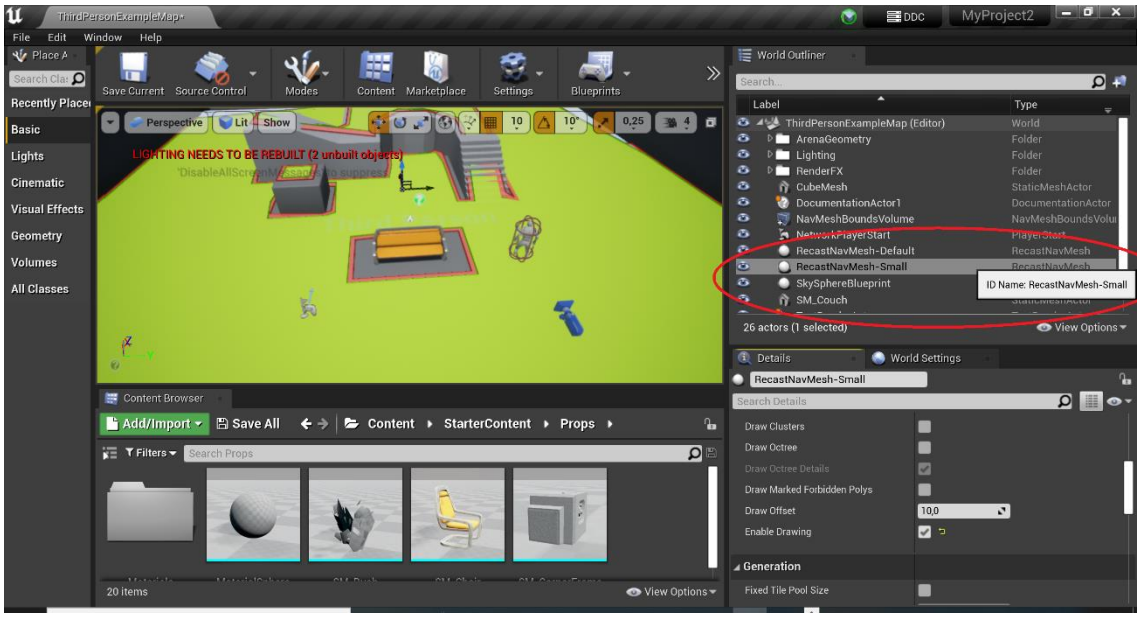
Creemos primero un agente que sea más pequeño y, por tanto, necesite una NavMesh distinta para poder navegar por todas las zonas habilitadas para su tamaño. Con copiar nuestro agente y escalarlo se recalculará la información para su Navegación en función del tamaño de su cápsula.



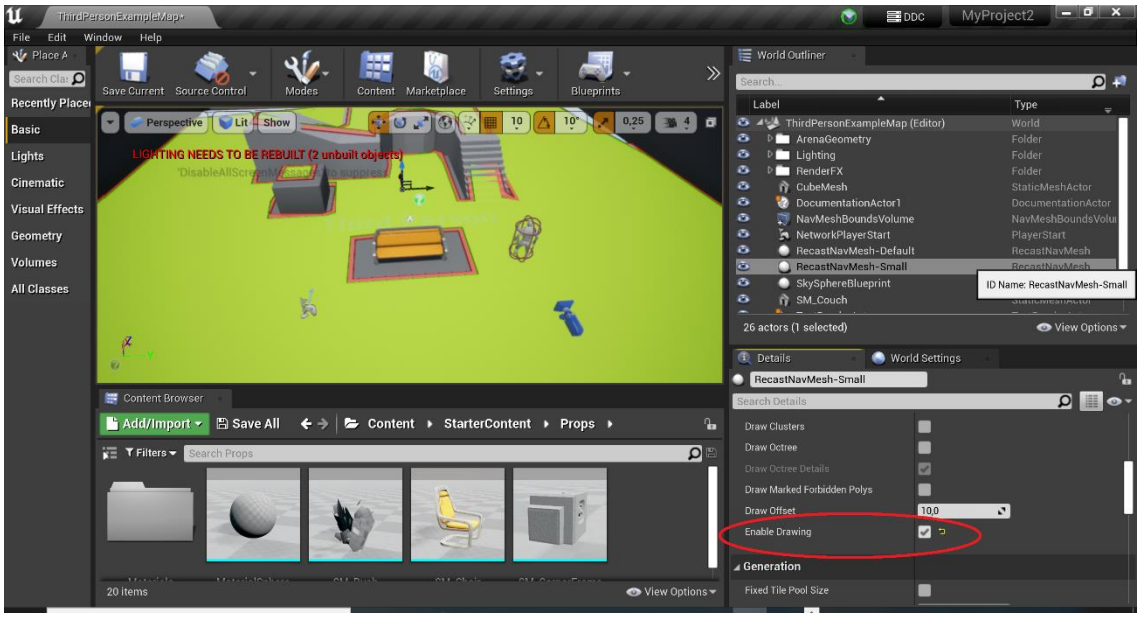
Para que este agente pueda tener una NavMesh adaptada a su tamaño, en la configuración del proyecto, en el apartado Engine - Navigation System, añadiremos nuevos elementos al array de Supported Agents, configurando cada uno Small según sus necesidades de navegación.



Veremos que en la escena, donde a partir de la creación del NavMeshVolume, ya se nos había creado un RecastNavMesh-Default anteriormente, ahora se nos creará uno nuevo, para el nuevo tipo de agente soportado que hemos creado, en este caso RecastNavMesh-Small.



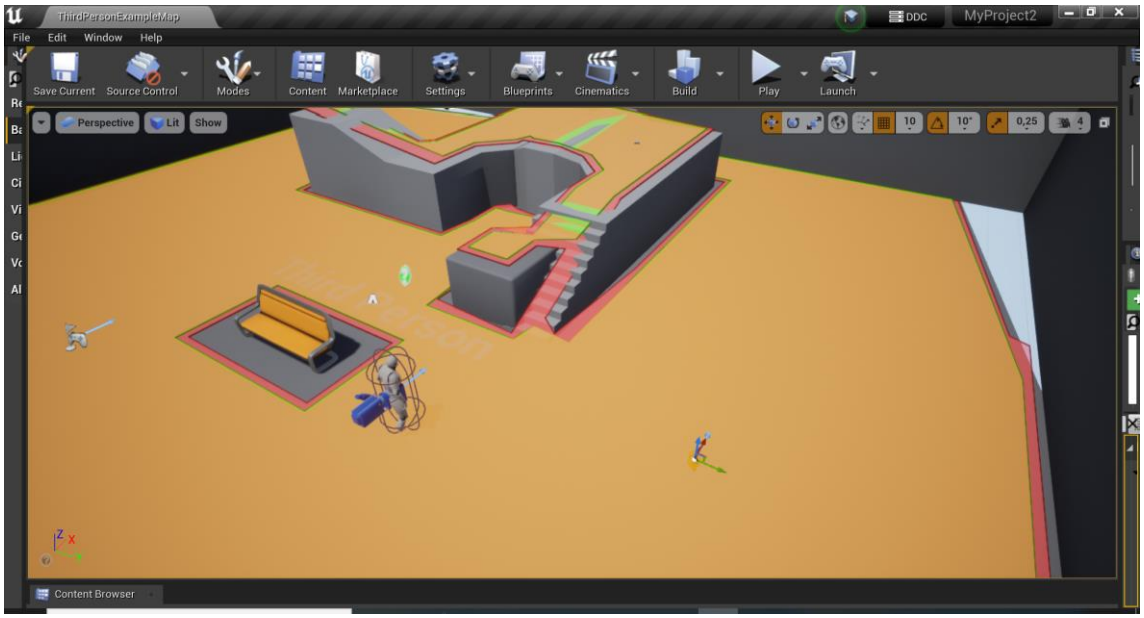
Para que se pinte la NavMesh generada para nuestro nuevo tipo de agente, debemos activar el CheckBox de Enable Drawing (parece que Unreal no guarda esta configuración de debug correctamente, por lo que deberemos activarlo cada vez que abramos el proyecto)



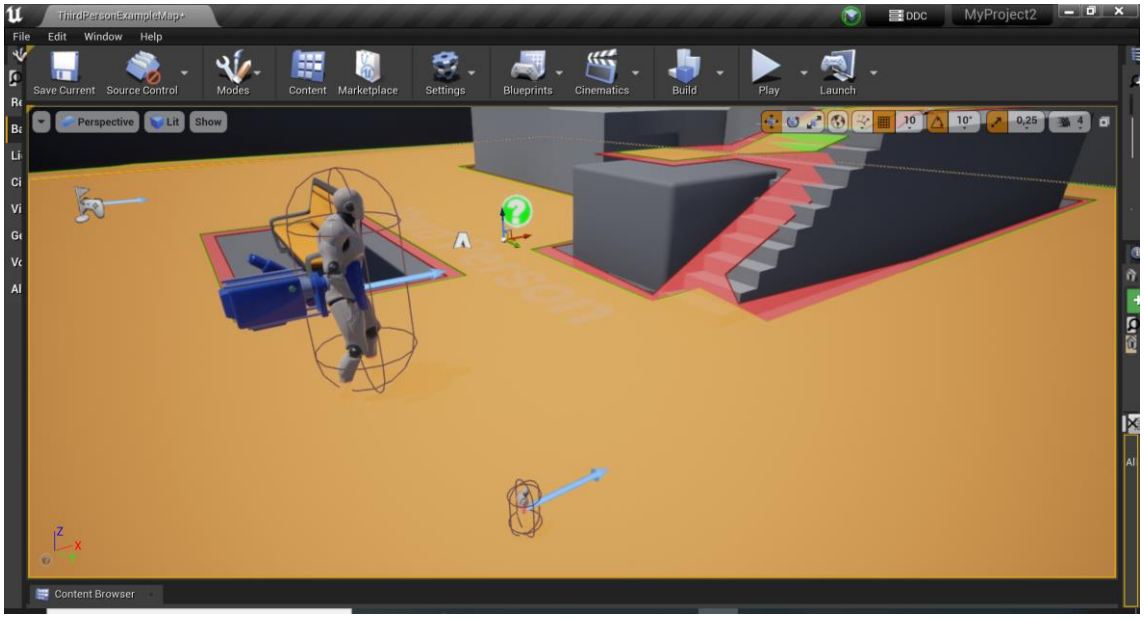
Una vez que ya tenemos nuestras dos NavMeshes configuradas y generadas a partir del mismo NavMeshBoundsVolume podremos comprobar, si por ejemplo adaptamos la escena para que nuestro nuevo agente pequeño tenga un camino estrecho por el que circular por el que el agente por defecto no quepa, cómo esa zona sólo será Navegable por el pequeño.

Ejercicio pathfinding

Hemos montado un obstáculo en las escaleras de subida que impida que un personaje de tamaño normal pueda acceder, pero nuestro mini-personaje sí tenga una NavMesh que suba hasta arriba.



Si mandamos al personaje de tamaño normal subir a un Waypoint situado arriba de las escaleras llegará hasta el punto de la NavMesh de la zona de abajo que más cerca le deje del punto inaccesible para él.



Pero si enviamos a nuestro pequeño mini-agente entonces sí podrá subir, al tener una NavMesh adaptada a su tamaño.