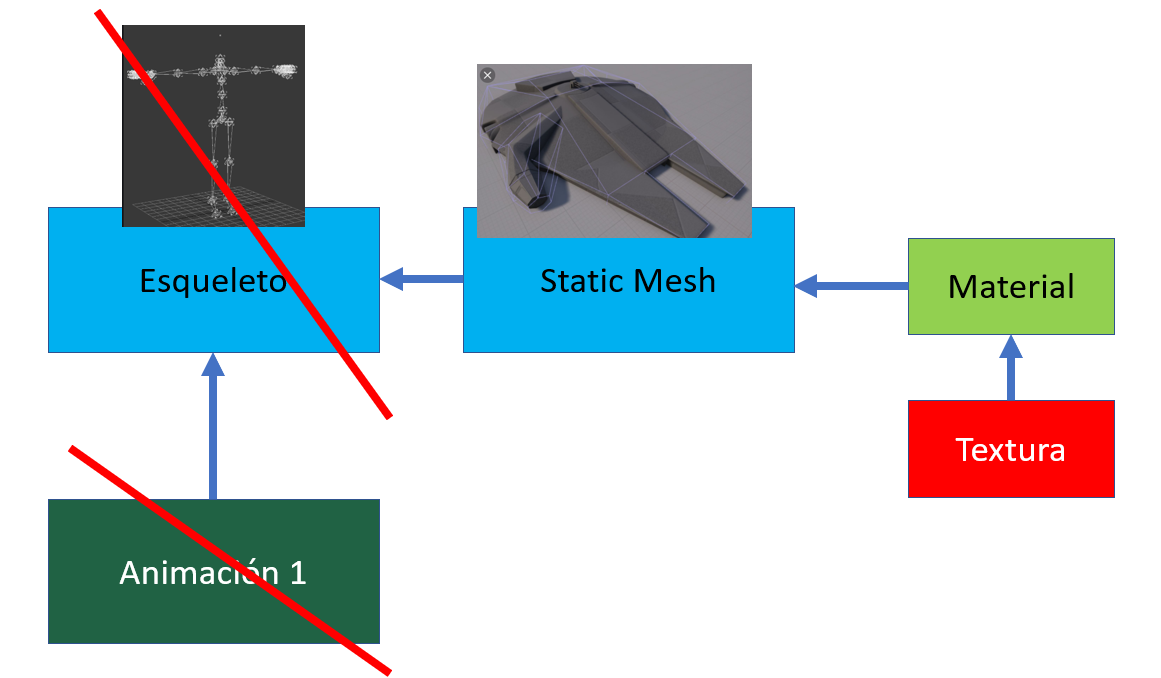
## Enemigo

Nos situamos en la Carpeta **Jugador**/**Graficos**  
  
Importamos el fichero **captus.obj.** Se trata de una malla estática sin esqueleto.

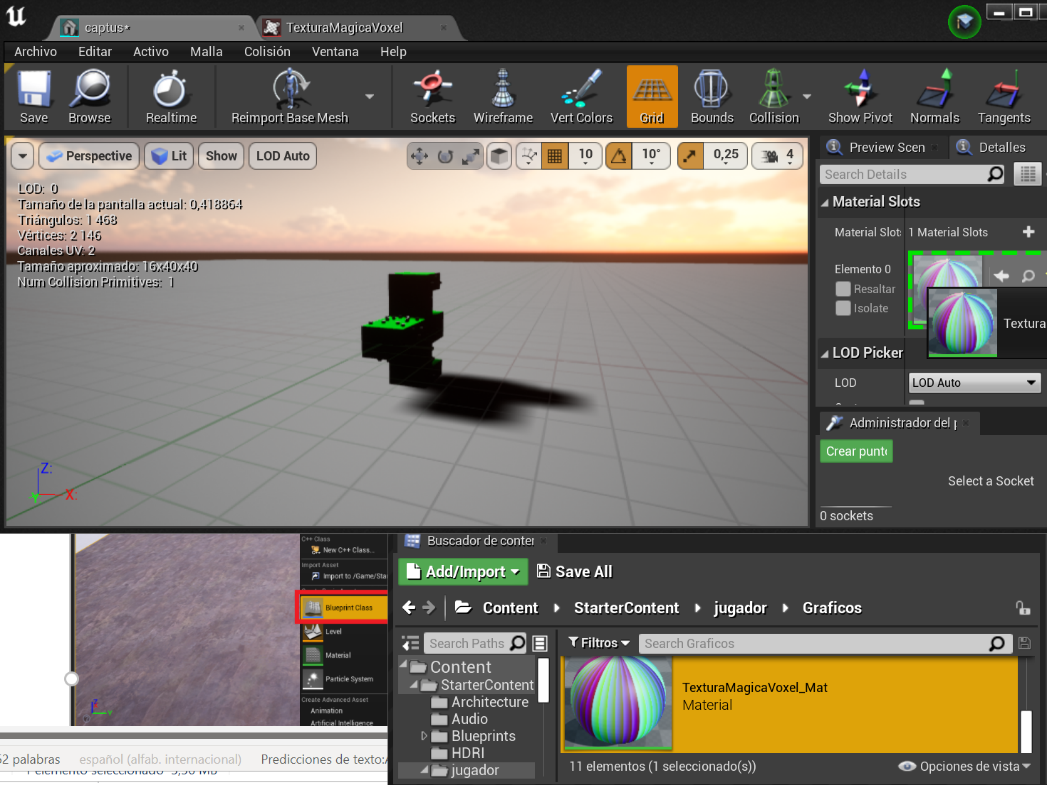


Las mallas estáticas, no tienen esqueleto interior, por lo tanto, no pueden tener animaciones.



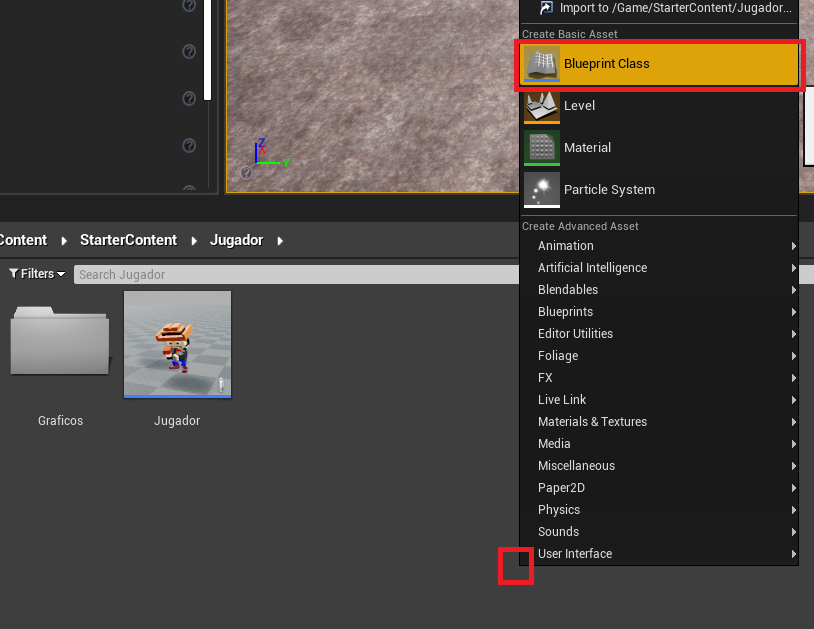
Abrimos el fichero del **captus**.

  
Establecemos la textura.

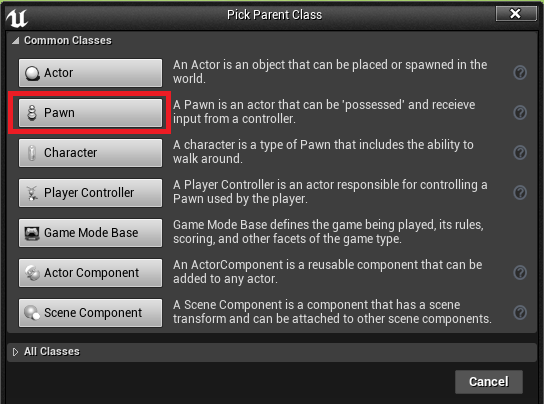


Volvemos a la carpeta **jugador**,

Pulsamos botón derecho sobre la superficie de la carpeta y creamos una nueva **Blueprint Class.**



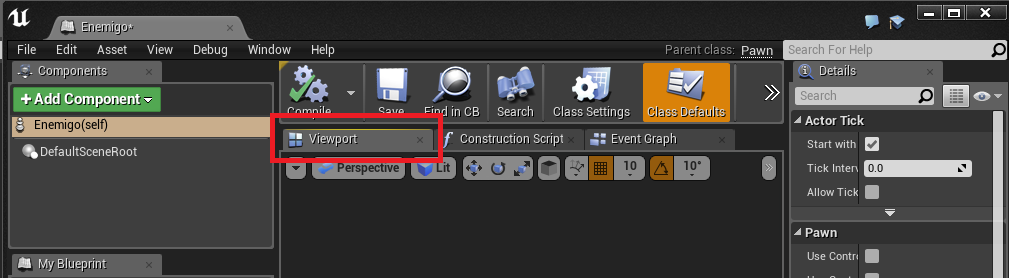
Elegiremos el tipo de clase Pawn (es más simple que Character, trae menos componentes inicialmente agregados)



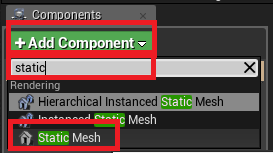
Cambiamos el nombre a **Enemigo**.



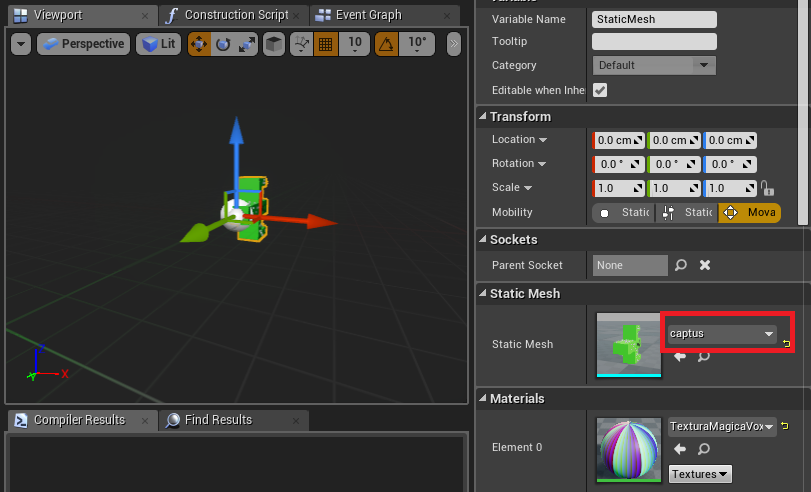
Hacemos doble click para poder editar el contenido de la clase, comenzamos por la pestaña del **Viewport**.



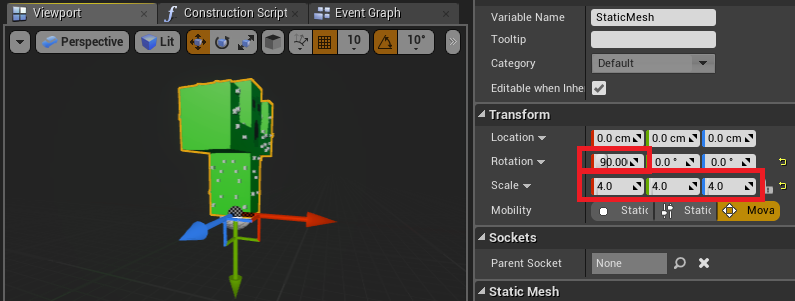
El enemigo no tiene ningún componente, vamos a comenzar agregando uno de tipo **StaticMesh** (Pulsamos en **+ Add Component** para agregarlo)



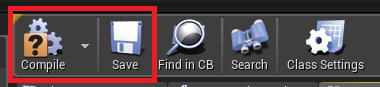
Seleccionamos el componente y editamos sus propiedades en la parte derecha, comenzando por la **Static Mesh**



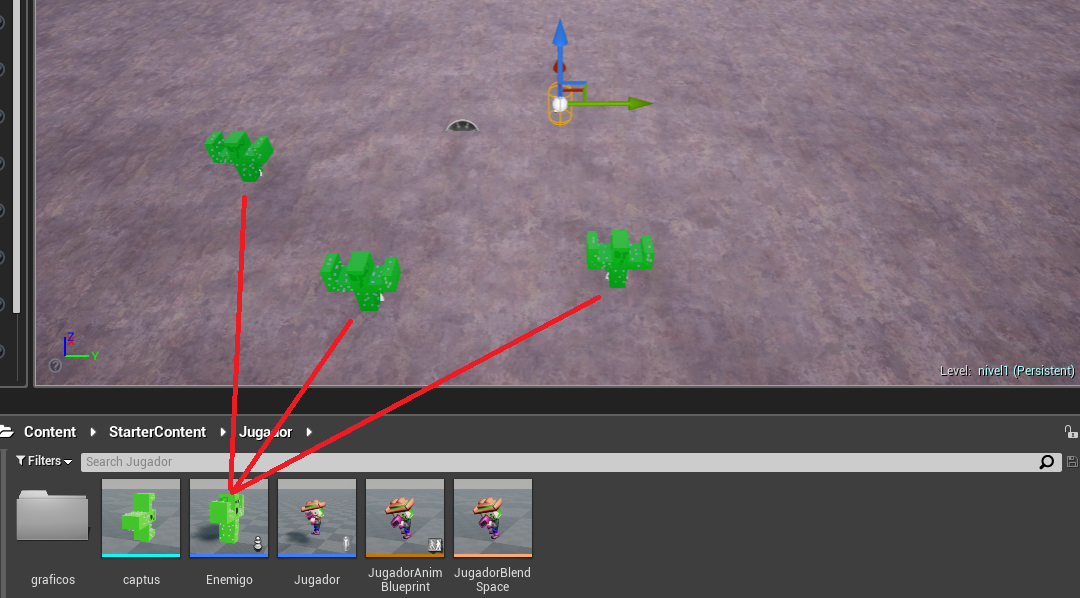
Cambiamos la rotación y el escalado.



Compilamos y guardamos los cambios.



Salimos al editor y arrastramos 3 enemigos cerca del componente **PlayerStart** (donde nace el jugador).

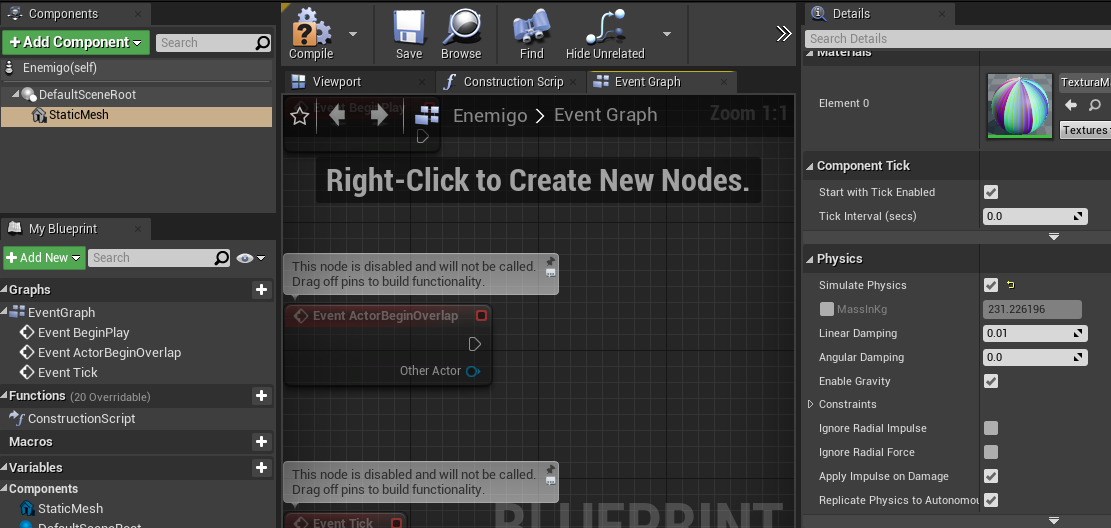


Probamos a ejecutar el juego.



Volvemos a entrar en el Enemigo.

Vamos a seleccionar su componente **Static Mesh** y modificar la propiedad **Simulate Physics.**



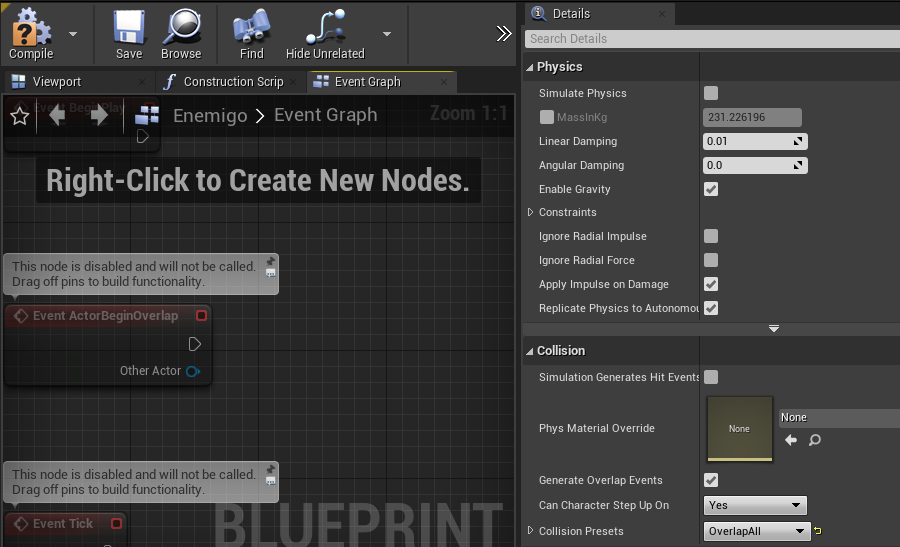
Guardamos los cambios y ejecutamos el juego, ahora los Enemigos tienen propiedades físicas.



Entramos de nuevo en la clase **Enemigo.**

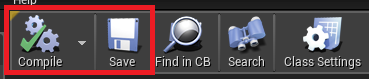


Desactivamos la casilla marcada anteriormente **Simulate Physics** y cambiamos el valor de propiedad **Collision Presents** por **OverlapAll** (Está es la propiedad principal que define el tipo de colisión)



**OverlapAll** no bloquea los elementos,son “etéreos” aunque es posible capturar el evento y ejecutar lógica cuando se produce la superposición.

Compilamos y salvamos los cambios.



Probamos el juego

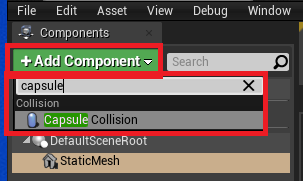


No bloquea físicamente.

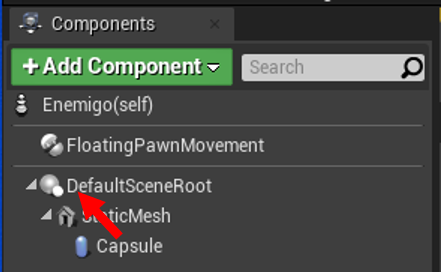
# Enemigo con componente collision

Este enemigo había sido creado de forma “demasiado simple” , vamos a utilizar un componente de tipo **collision** para que no use la **malla estática** como elemento principal (el uso de la malla estática complica el calculo de colisiones y también puede generar problemas en el método IA Move to que utilizaremos posteriormente).

Agregamos un nuevo componente de tipo **Capsule Collision**



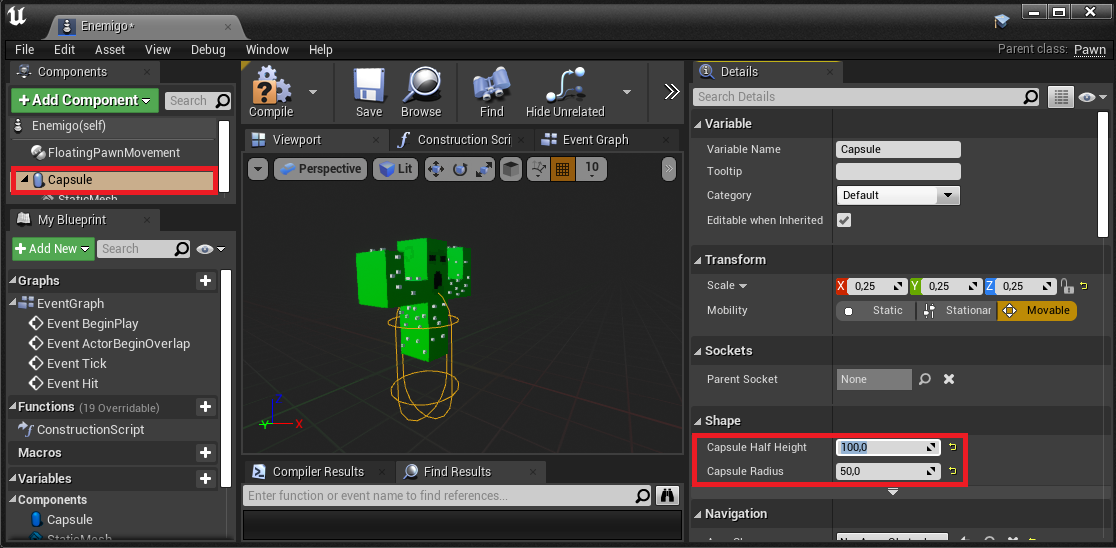
Una vez agregado tenemos que hacer que la capsula sea el **DefaultRoot**, es decir que la **StaticMesh** cuelgue de la capsula



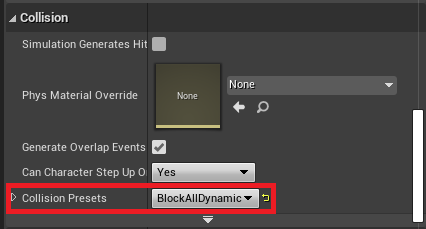
El resultado debería ser el siguiente



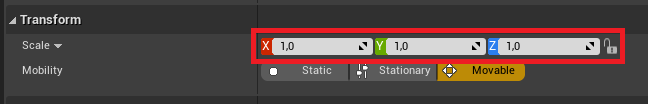
Seleccionamos la capusla y cambiamos sus dimensiones



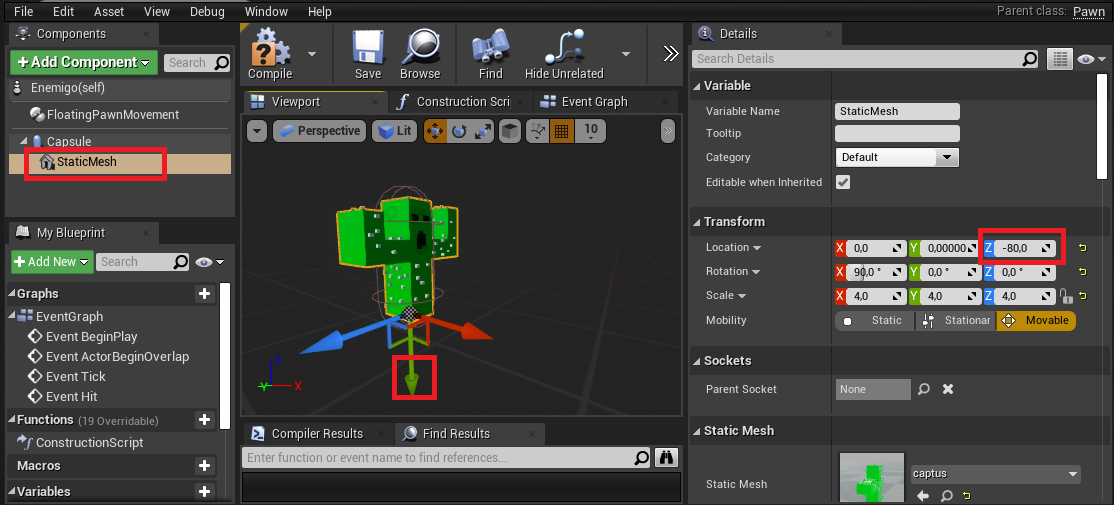
Vamos a cambiar el tipo de **Collision Presents** de la Capsula al valor **BlockAllDinamic** , para que genere choques reales en lugar de overlaps (hit).



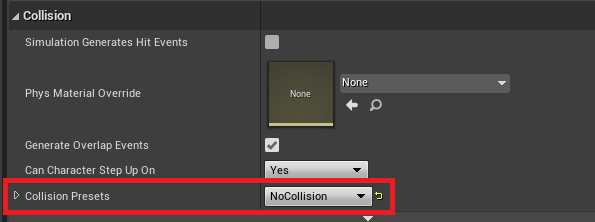
Debemos asegurarnos de que el **escalado** de la capsula está a 1, a veces, a veces Unreal lo cambia de forma automática si arrastramos componentes a otras posiciones



Ahora seleccionamos la **StaticMesh** y la movemos para “encajarlo” en la capsula, la capsula siempre debe sobresalir.

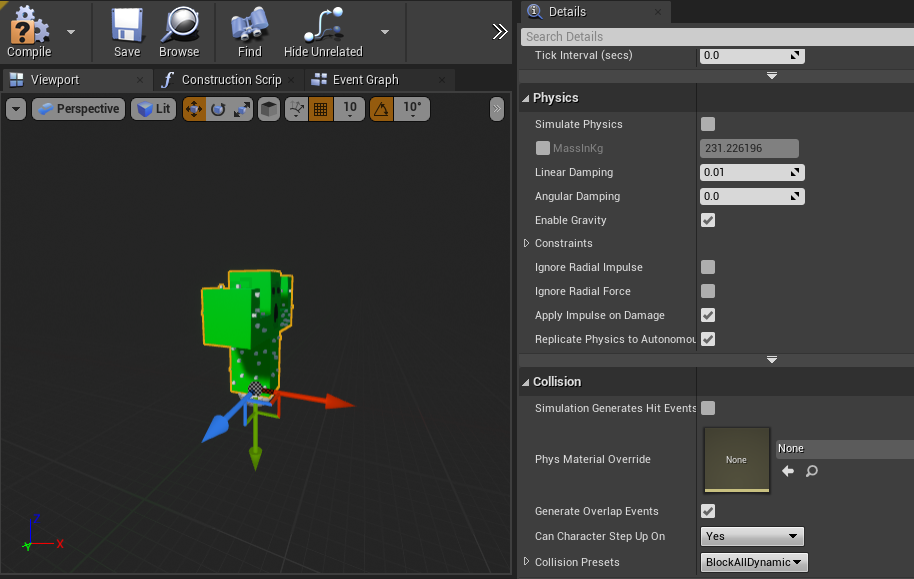


Si quisiéramos (aunque no es totalmente necesario) podríamos desactivar las colisiones de la **StaticMesh**, ya que se esta delegando en su elemento padre (el capsule collision)

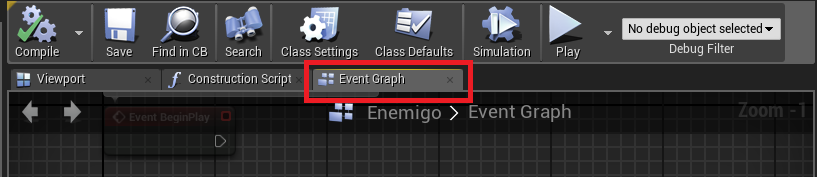


# Enemigo que desaparece al golpearle

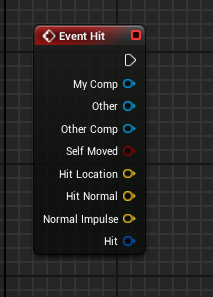
Volvemos a abrir la clase Enemigo y colocamos la propiedad **Collision Presents** a **Block All Dynamic**



Abrimos la pestaña del **Event** **Graph** para programar la lógica.



Buscamos una superficie libre y creamos un nuevo evento **Event Hit.** Este evento se ejecuta automáticamente cada vez que se produce un choque, nos da mucha información sobre el choque. La variable más interesante es **Other**, contiene información sobre el actor con el que ha chocado el Enemigo.



Vamos a sacar un flujo de la variable **Other** y ver si se trata del Jugador, haciendo un **Cast to Jugador** (es la forma que tiene unreal de comprobar el tipo de una variable, no pasa nada si fallan).

|  |  |
| --- | --- |
|  |  |

El **Cast to Jugador** tiene dos salidas, una si realmente era el jugador y otra si no lo era, además si es el jugador retorna una referencia al jugador **As Jugador**. La función **Destroy Actor**, destruye cualquier actor. ¿Qué diferencia hay entre los siguientes dos Blueprints?

|  |  |
| --- | --- |
| Ejemplo 1  Self.DestroyActor(); | Ejemplo 2 Jugador.DestroyActor() |

Dejamos la opción Ejemplo 1, para que desaparezca el propio **Enemigo**.

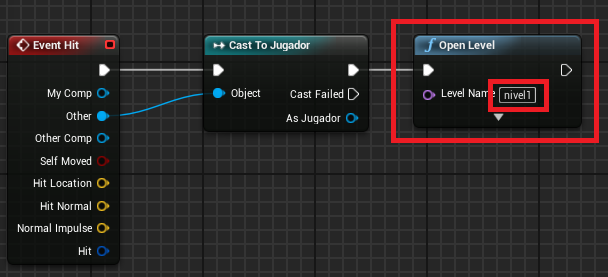
Compilamos y guardamos los cambios.



Probamos el juego, los enemigos deben desaparecer al golpearlos.



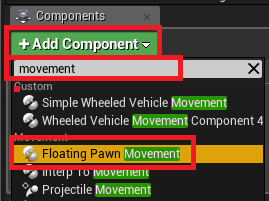
Sí quisiésemos que cuando un **Enemigo** golpease al **Jugador** se reiniciase el nivel podríamos utilizar la función **Open Level**, e introducir como parámetro el nombre del nivel, en este caso nivel1.



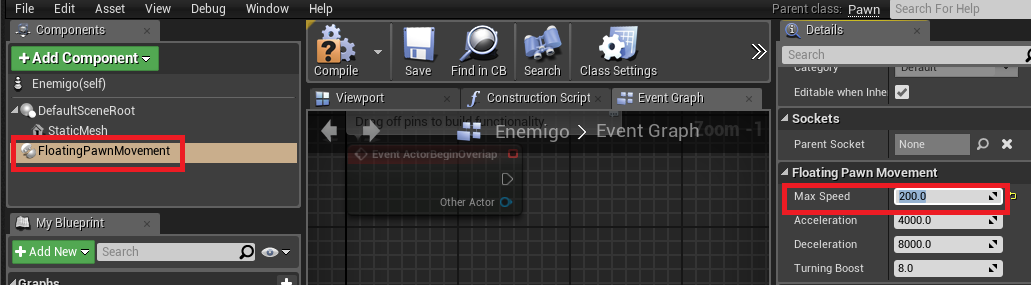
# Enemigo que se mueve

Abrimos la clase **Enemigo**.

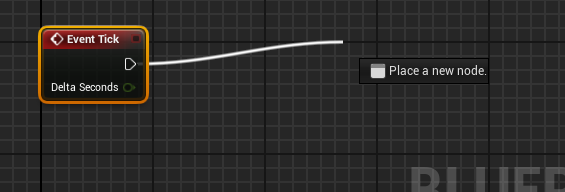
Debemos agregar un componente que le permita moverse, en este caso utilizaremos el **Floating Pawn Movement**, (movimiento simple para objetos flotantes).



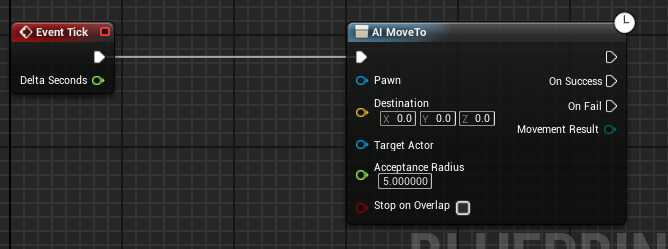
Seleccionamos el componente **Floating Pawn Movement**, en el menú de la izquierda editamos sus propiedades, concretamente la **Max Speed** , ponemos 200.



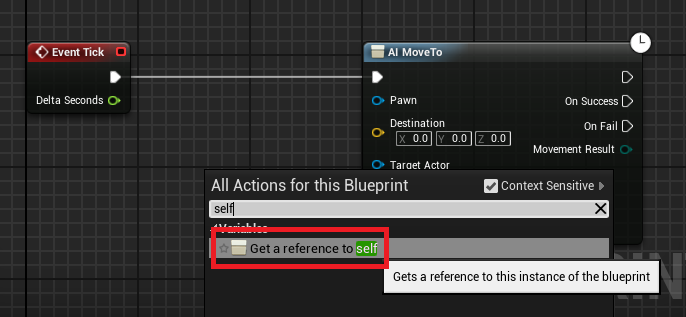
Vamos al **Event Graph** para continuar especificando código, localizamos la función **Event** **Tick** (Es una de las 3 que vienen pre-incluidas).



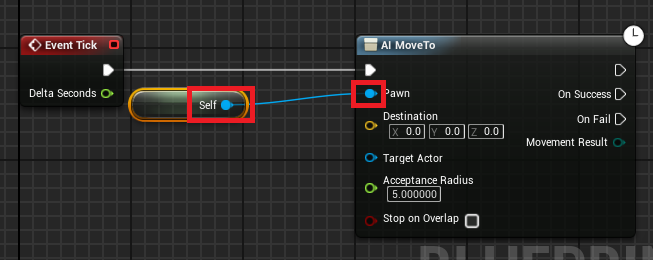
Llamamos a la función **AI MoveTo** para hacer que siga al jugador.



Está función recibe varios parámetros el primero de ellos es el pawn que va a efectuar el movimiento, obtenemos la variable **Get a Reference to Self**.



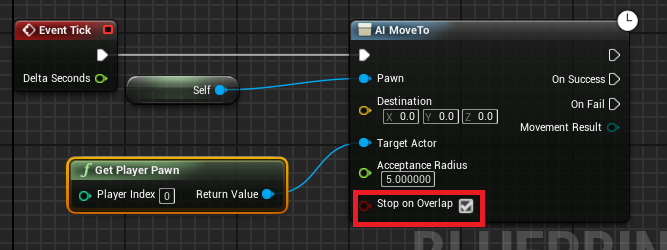
Conectamos el parámetro.



Ahora vamos a crear una variable para el **Target Actor** , actor al que debe seguir. Buscamos el parámetro **Get Player Pawn** (está función permite obtener una referencia al jugador en cualquier parte del código)



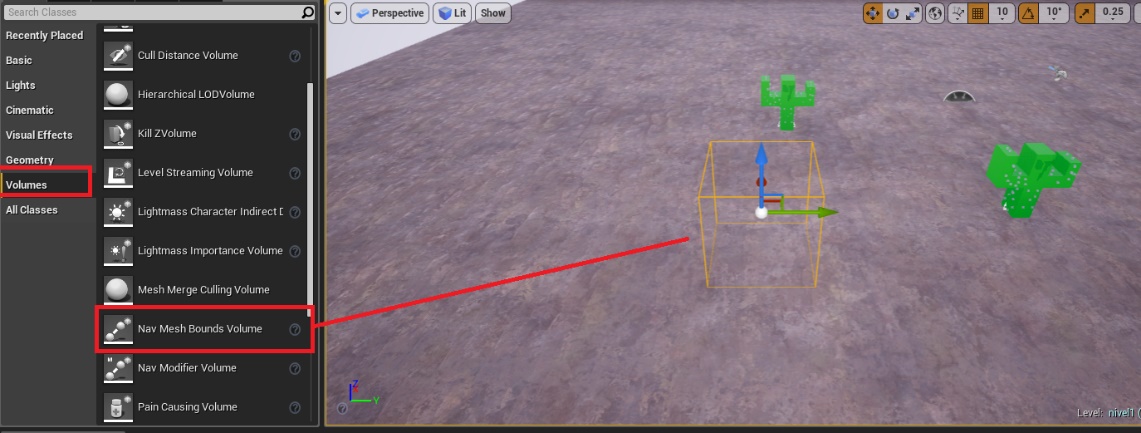
Por último, activamos la entrada boolean **Stop on Overlap**, para que se detenga una vez choca.



Compilamos y guardamos los cambios.



Volvemos al nivel, para que los enemigos nos sigan debemos generar una malla de navegación. Incluimos el elemento: **Volumes / Nav Mesh Bounds Volume**

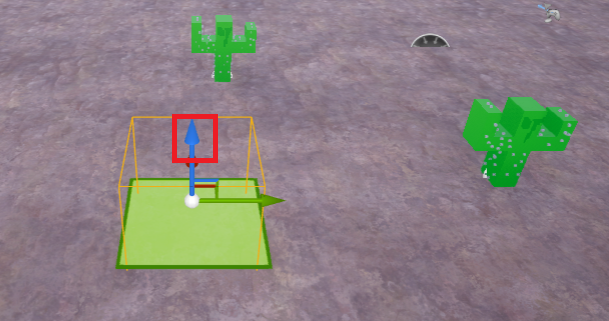


Este cuadrado no es visible en el juego, todo lo que corta se convierte en área de navegación para los enemigos.

Pulsamos la letra **P**, para que se remarquen las superficies cortadas.

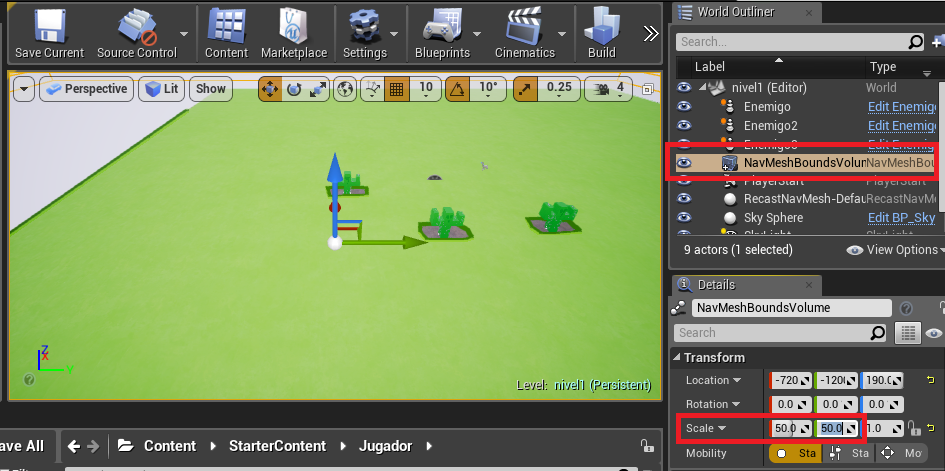


Sí no lo está lo “clavamos” ligeramente en el suelo.

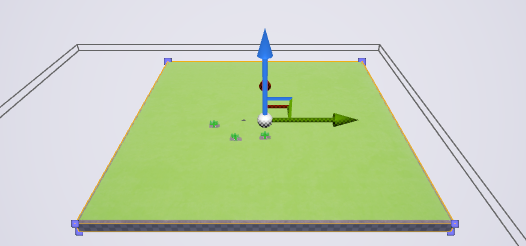


Ahora lo escalamos para que ocupe todo el mapa (no importa que se sea más grande que el mapa y se salga).

Nos aseguramos de tener seleccionado el **NavMeshBoundsVolumen** en la vista del **World Outliner.**

****

Nos aseguramos de que todo el suelo este cubierto de la malla de navegación.

****

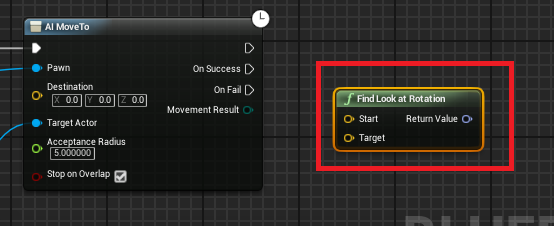
Probamos el juego, los enemigos se moverán hacia nosotros con velocidad 200.

****

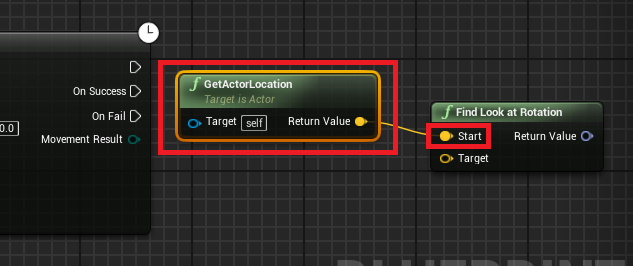
## Enemigo - Hacer que la malla rote hacia el jugador

Volvemos a la clase **Enemigo**, vamos a hacer que cada iteración del **Event Tick**, el enemigo rote hacia el jugador.

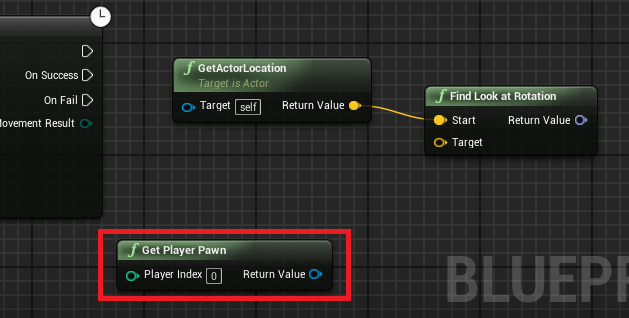
Agregamos la función **Find Look at Rotation** (recibe dos localizaciones y calcula la rotación que hay dar a la primera para que mire hacia la segunda).



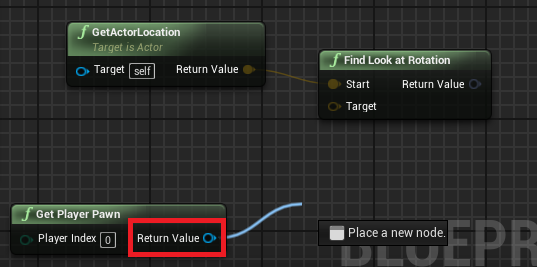
El primer parámetro **Start**, será la posición del propio Enemigo, se obtiene con **GetActorLocation**.

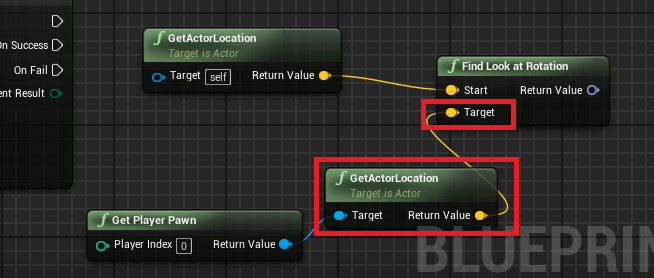


La segunda **Target**, es la localización del jugador, como vimos previamente se obtenía con **Get Player Pawn.**

****

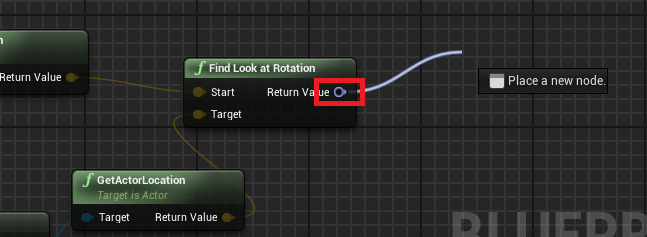
**Ger Player Pawn** retorna una referencia al jugador principal**,** a partir de la referencia podemos obtener el **GetActorLocation** del jugador

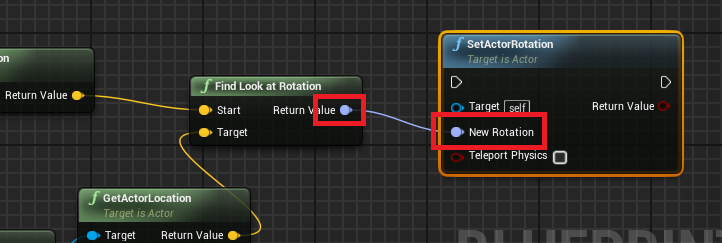
****

****

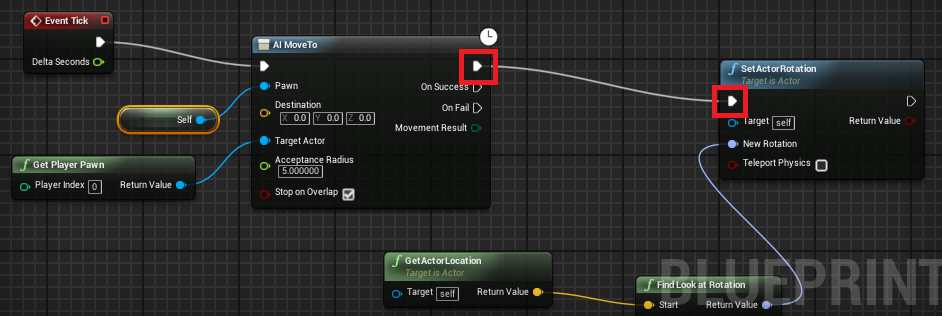
Partimos de la salida de **Find Look at Rotation** y buscamos la función Set Rotation (para colocar esa rotación al propio Enemigo).

Buscamos la función **SetActorRotation**.

****

****

Nos aseguramos de conectar el flujo de ejecución.

****

Compilamos y guardamos los cambios

****

Probamos el juego, los enemigos rotan hacia el jugador.



Conceptos, **Transform** contiene toda la información sobre la colocación de un elemento en el mapa (**localización, rotación y escalado**).

