UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería de Computadores

Trabajo Fin de Grado

Planificación de trayectorias 3D para agrupaciones de cuerpos no controlables en videojuegos.

UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería de Computadores

Trabajo Fin de Grado

Planificación de trayectorias 3D para agrupaciones de cuerpos no controlables en videojuegos.

Autor: Javier Ballesteros Morón

Tutor: Antonio Manuel Silva Luengo

ÍNDICE GENERAL DE CONTENIDOS

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1. RESUMEN

El objetivo del trabajo es lograr que una formación de cuerpos voladores se desplace a unos objetivos ubicados en un escenario 3D. La formación desconoce completamente el escenario, por lo que es un algoritmo de búsqueda de caminos a ciegas.

La formación funciona en torno a un líder y se deforma al detectar obstáculos, para volver a formar tras sortearlos.

El algoritmo de trayectoria empleado es un algoritmo propio nacido de una aproximación tridimensional a un algoritmo Bug para búsqueda de caminos en un espacio bidimensional.

The goal of this project is to achieve that a formation composed of flying bodies manages to move to multiple targets located in a 3D map.

The formation knows nothing about the map, so it's a blind pathfinding algorithm.

The formation works around a leader and it gets deformed whenever an obstacle is detected. The flying bodies form back after they pass the obstacle.

The algorithm implemented is my own idea born from a tridimensional approximation to a bug algorithm for blind pathfinding in a bidimensional space.

2. PLANTEAMIENTO DEL PROBLEMA Y OBJETIVOS

2.1. Planteamiento

Partimos desde un conjunto de puntos repartidos por un escenario tridimensional, a los cuales tiene que llegar una formación de cuerpos voladores sin conocer nada más que las coordenadas donde se ubican. Los puntos los llamaremos "targets".

El escenario es desconocido para los cuerpos voladores.

La formación se deforma cuando encuentra un obstáculo y vuelve a formarse de nuevo una vez sorteado.

No se pueden usar los sistemas de navegación ya implementados por la herramienta elegida (Unreal Engine 4).

Los cuerpos voladores tendrán sensores que les permiten detectar obstáculos próximos.

2.2. Objetivos

La formación de cuerpos voladores **debe alcanzar los targets** sorteando los obstáculos que se puedan encontrar por el camino. La ruta no tiene por qué ser la óptima, pues al ser un entorno desconocido puede incluso no haber solución.

Los cuerpos voladores deben volar en una formación que se deforme al detectar obstáculos para sortearlos con mayor eficiencia. Esto implica una posterior reagrupación de la formación al sortear el obstáculo.

3. ANTECEDENTES / ESTADO DEL ARTE

Existen muy pocas implementaciones de algoritmos de búsqueda de caminos en espacios tridimensionales, por lo que he basado ampliamente el algoritmo del proyecto en un algoritmo bug.

La mayoría de las búsquedas de caminos en espacios tridimensionales emplean un algoritmo A*, pues otorga siempre una solución muy eficiente (si existe), pero implica implementar una grid tridimensional que puede conllevar costes de computación muy elevados si el escenario crece en tamaño.

Algoritmo bug

Se trata de un algoritmo de búsqueda de caminos a ciegas en el que se parte de la base de que un robot equipado con sensores está ubicado en un plano 2D. Hay una posición inicial y un objetivo definidos y el robot siempre conoce la distancia restante hasta llegar al objetivo.

Cuando el robot llega a una pared u otro obstáculo, lo detecta con los sensores y empieza a moverse alrededor del obstáculo. La imagen siguiente muestra diferentes aproximaciones del algoritmo bug, siendo algunas más precisas que otras.

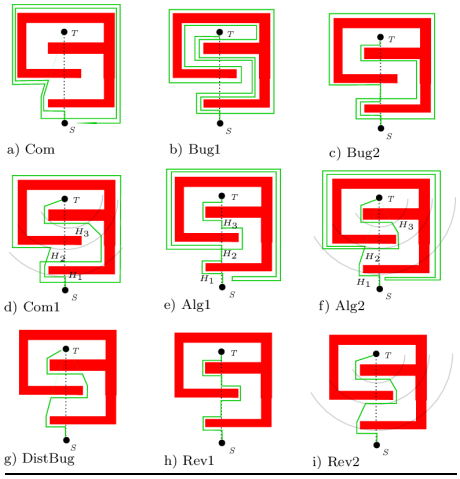

Ilustración 1 Algoritmo Bug

4. PLANIFICACIÓN

Tarea	Descripción	Periodo	Duración
Concepto para trabajar	Pensar en una idea para el trabajo de fin de grado. Se descartaron muchas.	Octubre de 2018 a marzo de 2019	20h
Búsqueda de algoritmos de búsqueda de caminos	Búsqueda de algoritmos de búsqueda de caminos tanto heurísticos como no informados y estudio sobre su posible viabilidad en un espacio tridimensional	14/06/19 – 30/06/19	18h
Búsqueda de información sobre formaciones de cuerpos	Búsqueda de información sobre formaciones y cómo implementarlas	14/06/19 – 30/06/19	12h
Aprendizaje de Unreal Engine 4	Aprendizaje de la herramienta de trabajo elegida. La parte más costosa del proyecto, pues dispone de una API inmensa y multitud de herramientas. Hice un curso entero de Unreal Engine 4.	01/07/19 - 31/07/19	110h
Prototipado en Blueprints	Implementación rápida de la mayoría de los algoritmos para disponer de un prototipo funcional y un depurado rápido y sencillo.	15/08/19 – 22/08/19	25h
Implementación de los árboles de comportamiento	Implementación de los árboles de comportamiento en los aviones para manejar su lógica para el algoritmo de búsqueda. Se implementaron con blueprints y con C++	15/08/19 – 31/08/19	10h

Implementación de los sensores en C++	Sólo implementé los sensores en C++. Opté por emplear 2 arrays de lásers, uno horizontal y otro vertical que detectan obstáculos y devuelven información sobre el obstáculo más cercano.	15/08/19 – 17/08/19	10h
Implementación de un algoritmo propio de búsqueda de caminos a ciegas.	El algoritmo en realidad consiste en tareas enlazadas mediante un árbol de comportamiento, por lo que excluiré el tiempo de implementación de los árboles de comportamiento.	22/08/19 – 31/08/19	20h
Implementación de formaciones	Las formaciones fueron implementadas al final. Opté por no incluir posibles colisiones entre aviones, debido a la enorme complejidad de adaptar algoritmos como RVO en un espacio tridimensional	31/08/19 – 4/09/19	20h
Documentación	La documentación externa la redacté al concluir el desarrollo del proyecto. El tiempo de la documentación interna va incluido en las etapas de implementación, pues lo hice al mismo tiempo.	20/10/19 – 5/11/19	30h
		TOTAL:	275h

5. IMPLEMENTACIÓN Y DESARROLLO

5.1. Herramientas elegidas

Para mi implementación del proyecto decidí emplear un motor gráfico, concretamente **Unreal Engine 4**. Para programar con Unreal Engine hay que usar **C++**. Como IDE para C++ he utilizado **Visual Studio 2019**.

Ilustración 2: Unreal Engine 4

Ilustración 3: Visual Studio 2019

¿Por qué Unreal Engine 4?

Debido a la necesidad de crear un entorno 3D, un motor gráfico facilita mucho la tarea. Unreal Engine es el motor gráfico más utilizado en la industria del videojuego. También se utiliza en arquitectura y cine. Es una herramienta muy compleja con una inmensa API. A pesar de que hay herramientas más sencillas como Unity, me decanté por Unreal Engine por mi familiaridad con C++ (Unity utiliza C# y javascript).

La otra razón que me hizo elegir Unreal Engine fueron sus árboles de comportamiento, que me parecían muy interesantes para la implementación del problema.

Para el control de versiones he utilizado GitHub por la familiaridad que tengo con él tras años usándolo en la universidad y porque para pequeños grupos o una sola persona funciona muy bien.

Lo ideal para Unreal Engine sería Perforce debido a la facilidad que aporta para trabajar con ficheros binarios (Blueprints), pero al ser un proyecto individual, no se aprovecha la utilidad.

5.2. Prototipado

Para sentar las bases del proyecto, lo primero es montar un escenario 3D. Unreal Engine trae varias plantillas para comenzar proyectos, por lo que utilicé la plantilla "Flying", que incluye un escenario con algunos obstáculos y un avión con un controlador en C++ que recibe inputs del teclado para moverlo. Esto último lo eliminé, pues los aviones no van a ser controlables.

Ilustración 4:Creación de proyecto y selección de plantilla

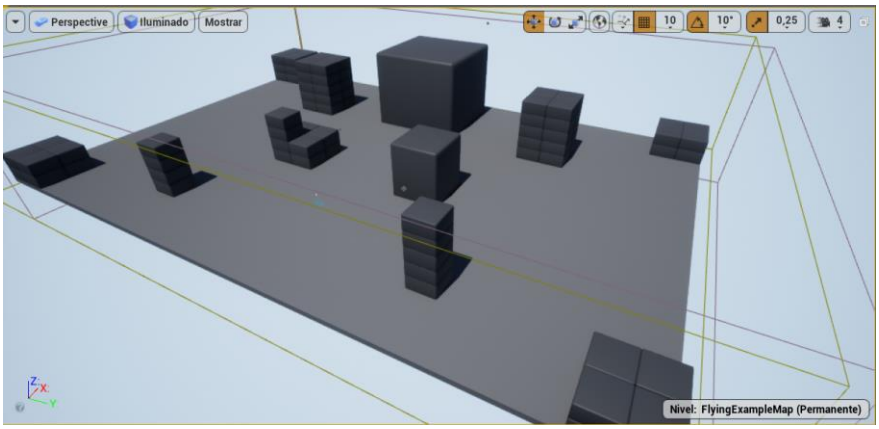

Ilustración 5: Mapa generado por la plantilla

Una vez generado el escenario, estructuré el proyecto en pequeñas tareas, siendo unas de mayor prioridad que otras:

- **Crear los sensores de los aviones.** (Alta prioridad, sin sensores no hay detección de obstáculos)
- **Movimiento** (Simplemente moverse directamente hacia el objetivo, alta prioridad)
- **Árboles de comportamiento** (el corazón del algoritmo, prioridad media)
- **Algoritmo para evitar obstáculos** (prioridad media)
- **Creación de formaciones** (Prioridad baja)
- **Formación deformable** (Prioridad baja)

Unreal Engine 4 dispone de Blueprints. Las Blueprints son una abstracción de C++ que permiten programar simplemente uniendo bloques de código. Tienen unas ventajas muy claras: Compilación muy rápida para poder probar el programa sin tener que esperar y visualización gráfica del flujo de ejecución, lo que permite encontrar fallos fácilmente.

Se puede descargar el prototipo del proyecto, implementado en Blueprints desde https://github.com/BlazeNeko/TFG/tree/e0df3fe3b784f76ab304a2fa775549158b944d45

Se trata de una versión antigua, por lo que no es completamente funcional, pero tiene un árbol de comportamiento con todo lo necesario para moverse entre puntos y esquivar obstáculos.

Si se va a probar el prototipo, habría que actualizar la versión de Unreal Engine del prototipo haciendo click en TFG.uproject > Switch Unreal Engine versión y seleccionando la versión que tengamos instalada.

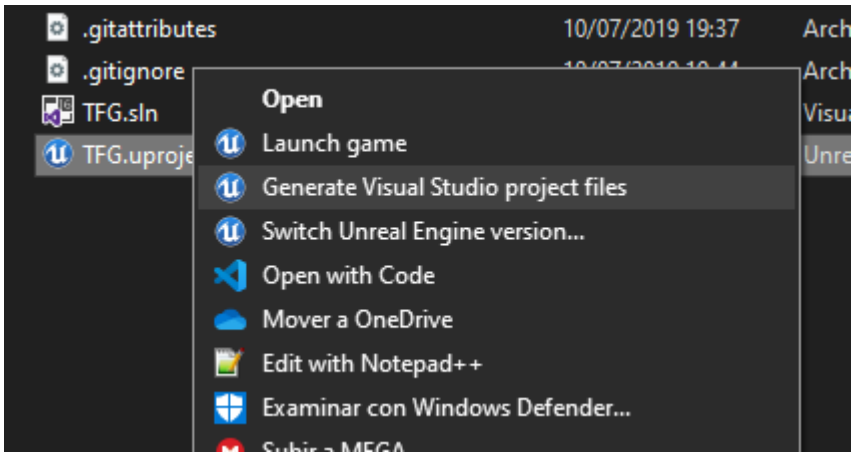

Ilustración 6:Actualizar versión del proyecto

Como pega, son menos eficientes que escribir el código, pero su rapidez de implementación y facilidad para depurar las hace ideales para implementar prototipos, que luego se pueden pasar a código.

Para el prototipado, el movimiento y el algoritmo los programé primero en Blueprints y posteriormente en C++.

En la carpeta del proyecto he añadido una carpeta llamada "Blueprint-Images" donde hay imágenes de todas las blueprints empleadas en el prototipo y el proyecto final. Esta es la única forma de poder verlas desde fuera del editor de Unreal Engine, ya que si no se verían como un fichero binario.

5.3. Sensores

Para la implementación de los sensores creé 2 arrays de lásers: uno vertical y otro horizontal. Los lásers cubren un abanico de 180º. El número de lásers y su alcance son ajustables y su ángulo se calcula mediante una interpolación entre los lásers de los extremos en función del número de lásers que haya.

Aparte de los 2 arrays de lásers, todos los aviones poseen un sensor en forma de caja que permite detectar únicamente los obstáculos ubicados en frente.

Inicialmente sólo está activo el sensor con forma de caja. Cuando se encuentra un obstáculo, únicamente el líder emite los rayos horizontales y verticales hasta que se rodea el obstáculo.

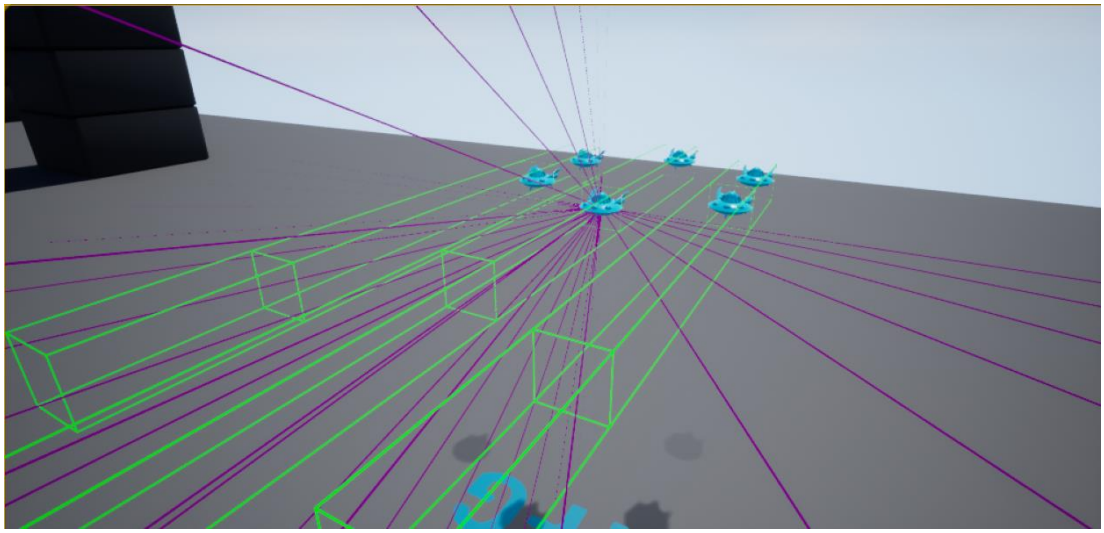

Ilustración 7:Sensores de los aviones

5.4. Movimiento

El movimiento de los aviones consta de dos partes: rotación y desplazamiento.

- **Rotación**

  Los aviones deben rotar hacia el objetivo. Es importantísimo, ya que, si los aviones no rotan, los sensores apuntarían siempre en la misma dirección.

  Para ello simplemente se calcula una interpolación entre la rotación actual del avión y la rotación que tenemos como objetivo en función de una velocidad de rotación indicada. Cada

tick del reloj se actualiza la rotación poco a poco para conseguir que gire gradualmente.

- **Movimiento**

  Los aviones están siempre en movimiento. Si es necesaria una rotación los aviones no paran, sino que rotan mientras se mueven.

  Unreal Engine facilta mucho el movimiento, ya que incluye funciones relacionadas con ello.

  Lo único que hay que hacer es calcular la dirección del movimiento y la velocidad del avión para emplear la función de movimiento de Unreal.

Los aviones se mueven hacia objetivos colocados previamente en el mapa.

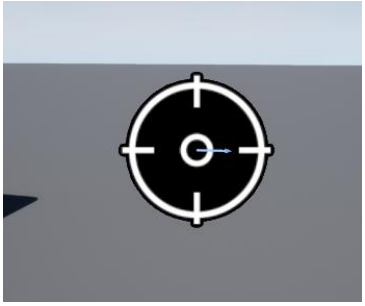

Ilustración 8:Objetivos de los aviones

5.5. Formaciones

Para crear la formación creé un generador de aviones. Esto me permite crear una formación circular con el número de aviones que especifique y el radio de la formación.

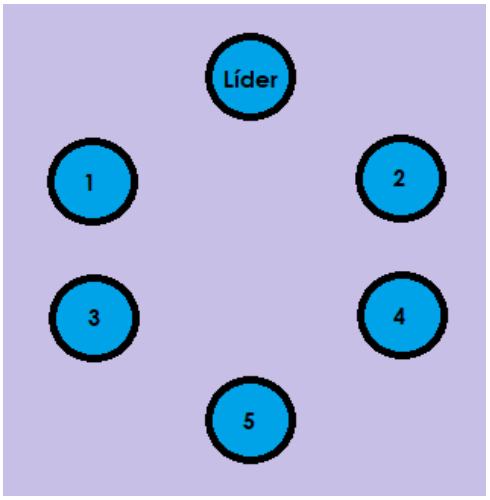

Ilustración 9:Ejemplo de formación de 6 aviones

Los aviones se añaden a una estructura de datos que contiene información sobre la formación (número en la formación y offset respecto al líder).

Cuando no hay obstáculos, se mueven según su offset respecto al líder.

Cuando se detecta un obstáculo, se mueven en fila respetando su número en la formación.

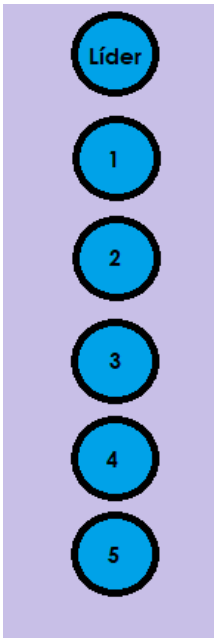

Ilustración 10:Formación sorteando un obstáculo

5.6. Árboles de Comportamiento

Los árboles de comportamiento permiten a los aviones ejecutar tareas implementadas en C++ o Blueprints. Inicialmente las programé en Blueprints, pero más adelante en C++.

Las **tareas** programadas son:

- **Choose Target:** Selecciona el siguiente objetivo en la lista de objetivos.
- **Move to target (sólo líder de la formación):** El líder se rota hacia el objetivo y se mueve hacia él en línea recta.
- **Avoid obstacles (Sólo líder de la formación):** Si los sensores han detectado un obstáculo, los láser nos devuelven información sobre él, concretamente el vector normal de la superficie donde impactó el láser.

  Si el obstáculo ha sido detectado con el sensor horizontal, calculo un vector "arriba" respecto a la posición del avión y calculo su producto vectorial con la normal del obstáculo, que devuelve una dirección paralela a la superficie del obstáculo para que el avión la siga.

  Si el obstáculo ha sido detectado con el sensor vertical, calculo un vector "derecha" respecto a la posición del avión y aplico los mismos cálculos mencionados en el párrafo anterior.

  Con estos cálculos evito la posibilidad de que el producto vectorial se produzca entre 2 vectores cuyo ángulo sea de 180º (saldría 0 en el producto vectorial y se pararía el movimiento).
- **Formation Movement (para todos los aviones menos el líder):**

  Cada avión se mueve según su offset respecto al líder de la formación. Si un avión no se encuentra en su posición, aumenta su velocidad hasta que se posiciona. Si el líder rota, el offset también

rota, de manera que la formación no se mantiene completamente estática.

- **Formation Avoid Obstacles Movement (para todos los aviones menos el líder):** Los aviones se mueven en fila porque se ha detectado un obstáculo previamente. Si se encuentran fuera de su posición en la fila aumentan su velocidad.

Hay dos árboles de comportamiento. Uno para el líder y otro para el resto de aviones.

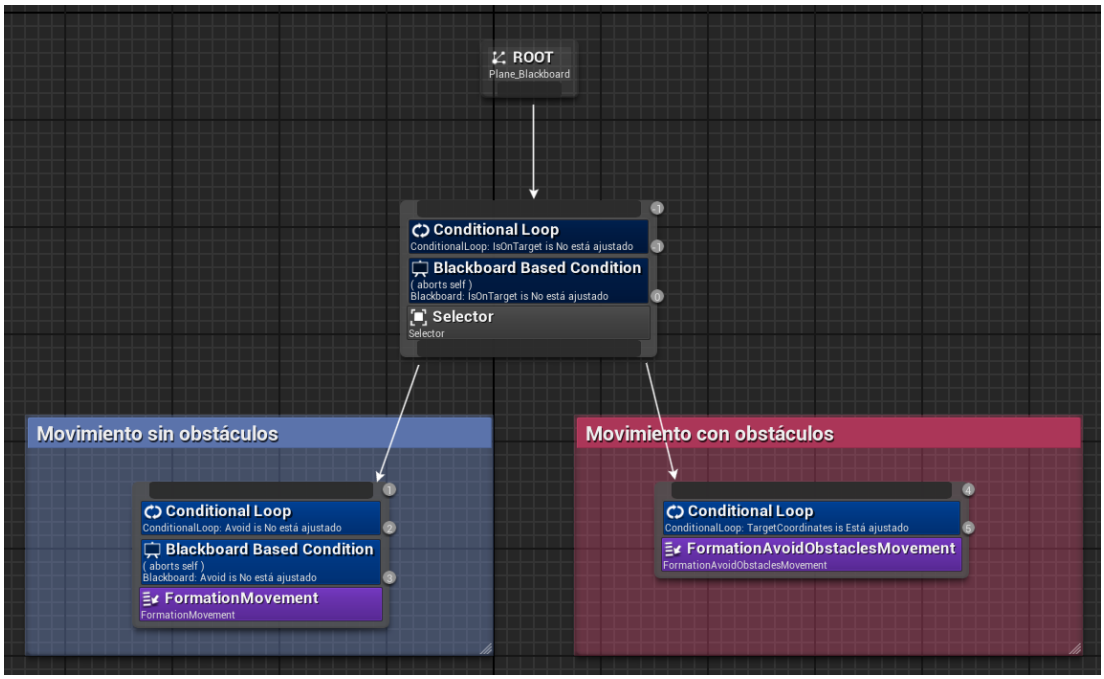

Ilustración 11:Ejemplo de árbol de comportamiento

Los algoritmos los implementé como un conjunto de tareas a ejecutar por los árboles, de tal manera que realizan lo siguiente:

a) Árbol del líder

a.1.) Si el líder se encuentra en el objetivo, elige el siguiente objetivo de la lista.

a.2.) Mientras el líder no esté en el objetivo, el líder **comprobará si tiene que evitar algún obstáculo**. Si no hay ninguno, ejecuta la tarea "**Move to target**". Si se detecta un obstáculo ejecuta "**Avoid obstacles**".

b) Árbol del resto de aviones

b.1.) Si el líder no ha detectado obstáculos, ejecuta "**Formation Movement**"

b.2.) Si el líder ha detectado obstáculos, ejecuta "**Formation Avoid Obstacles Movement**"

6. RESULTADOS Y DISCUSIÓN

Las pruebas consisten en ejecutar el programa para ver si los aviones alcanzan los objetivos designados.

Junto a este documento se ha adjuntado un video de la simulación.

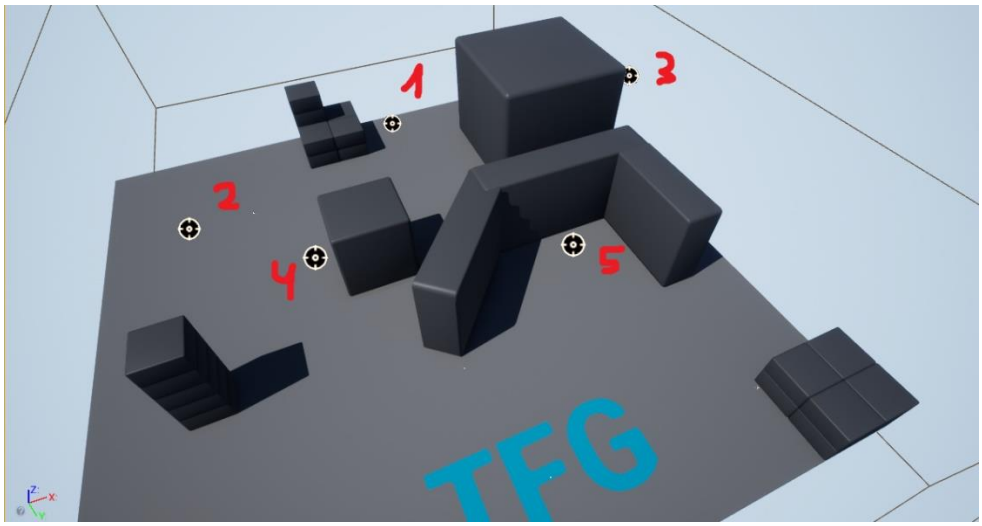

Ilustración 12:Escenario de pruebas

La imagen previa muestra el escenario de pruebas con 5 objetivos establecidos. La idea tras estos puntos era probar diferentes tipos de obstáculos (Esquinas, paredes, espacios cóncavos, zonas estrechas y zonas más amplias).

Los aviones aparecen próximos al punto 5, por lo que se podría considerar el punto de inicio también.

Para ir al punto 1, debe sortear una superficie con forma cóncava (el mayor problema que ha tenido la formación).

El punto 2 es simplemente un punto sin obstáculos para comprobar que los aviones forman correctamente.

Para ir al punto 3, en casi todas las ejecuciones los aviones han volado por el pasillo estrecho que se encuentra próximo, por lo que se puede ver cómo se comportan en espacios estrechos.

El propósito del punto 4 es que los aviones vuelvan por donde han venido. Esto al principio tuvo sus problemas porque uno o dos aviones se quedaban atascados en la esquina próxima al punto 3, pero conseguí solucionarlo ajustando parámetros como el radio de la formación o la velocidad a la que se mueven.

El punto 5 está pensado para que los aviones vuelvan cerca del punto de inicio y repitan todo de nuevo.

Las pruebas las he realizado en 2 equipos distintos:

- Un PC de sobremesa con 120 FPS estables. Los resultados en este PC han sido más satisfactorios debido a que los ticks han sido más rápidos y el delta time (tiempo entre frames) menor. El movimiento va en función del delta time para que los aviones se muevan a la misma velocidad en equipos de distinta potencia.
- Un portátil con 30FPS de media. Los resultados han sido peores (los aviones se quedaban atascados a veces, aunque conseguían salir de los apuros). Se debe a que al tener ticks más lentos a veces tardaba un poco más al detectar obstáculos.

Los resultados han sido satisfactorios en la gran mayoría de ejecuciones. Los aviones han conseguido volar hacia todos los puntos designados.

Las zonas conflictivas han sido los obstáculos entre los puntos 5 y 1 y la esquina próxima al punto 3 (este último completamente solucionado).

En algunas ocasiones los aviones dudaban por dónde girar en el punto 5, pero siempre han acabado saliendo del atasco.

7. MANUAL DE USUARIO

La carpeta del proyecto incluye una carpeta llamada Ejecutable. Abriendo el fichero ejecutable se puede ver una simulación del proyecto.

Requisitos: Windows 64 bits.

Los únicos controles para la simulación son las teclas W, A, S, D para poder mover la cámara por el escenario. La simulación consiste en los aviones moviéndose entre 5 puntos pre-programados (los de la imagen del apartado 6).

Para finalizar la simulación pulsamos Alt + F4.

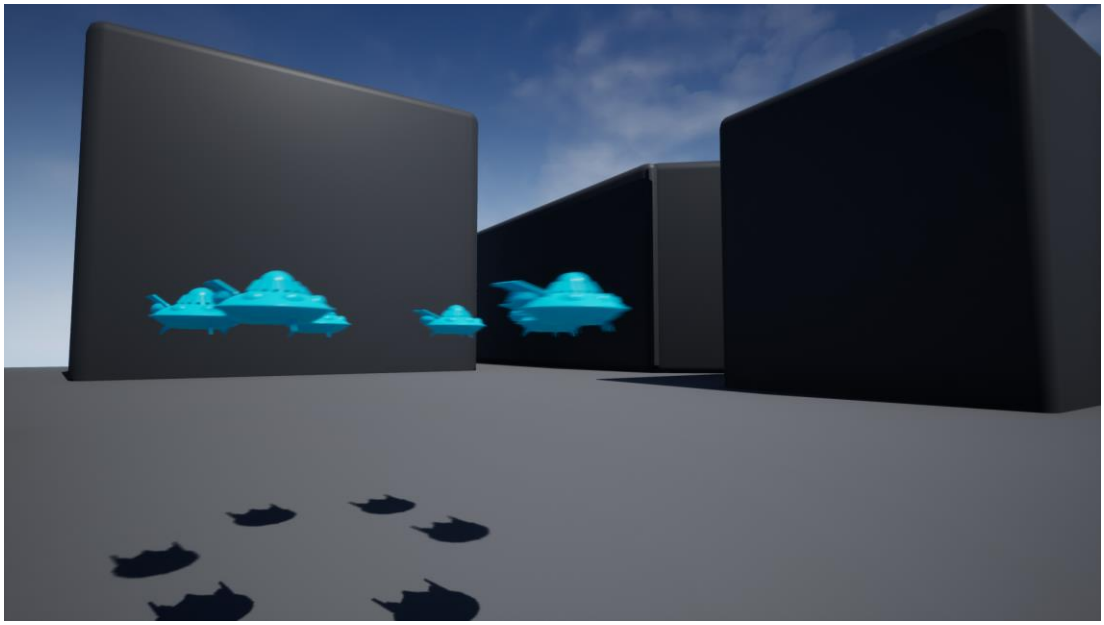

Ilustración 13:Captura de una ejecución del proyecto

La versión de usuario es un ejecutable externo al editor de Unreal Engine. No es configurable, pero desde el editor de Unreal Engine se pueden configurar parámetros tales como la velocidad o el radio de la formación.

8. INSTALACIÓN PARA PROGRAMADORES

8.1. Instalación de Unreal Engine y Visual Studio

Antes de nada. se requieren aproximadamente 25GB de espacio en el disco duro para Unreal Engine 4, otros 8GB para Visual Studio y otros 8GB para el proyecto (los ficheros generados por Unreal Engine 4 para el proyecto ocupan mucho espacio).

En la misma carpeta de la documentación vienen dos instaladores. Uno instala el Epic Games Launcher, necesario para instalar Unreal Engine. El otro instala Visual Studio.

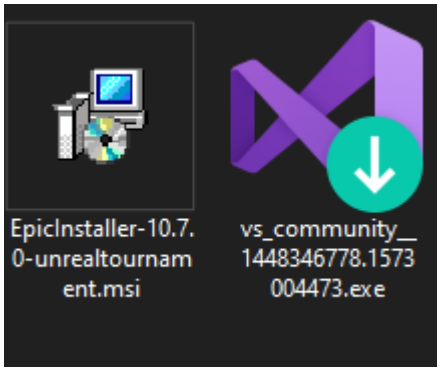

Ilustración 14:Instaladores

Una vez instalado el Epic Games Launcher hay que iniciar sesión o crearse una cuenta.

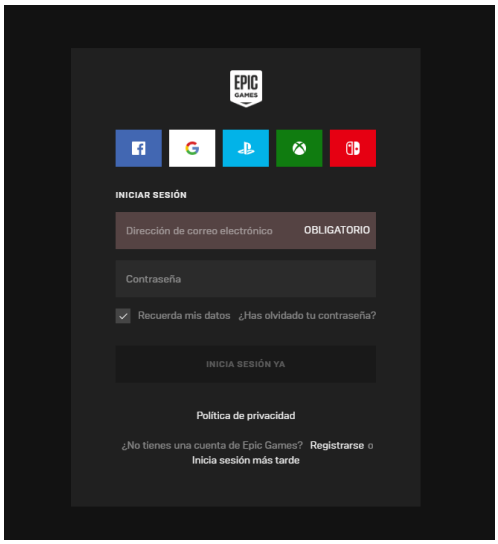

Ilustración 15:Log In Epic Games

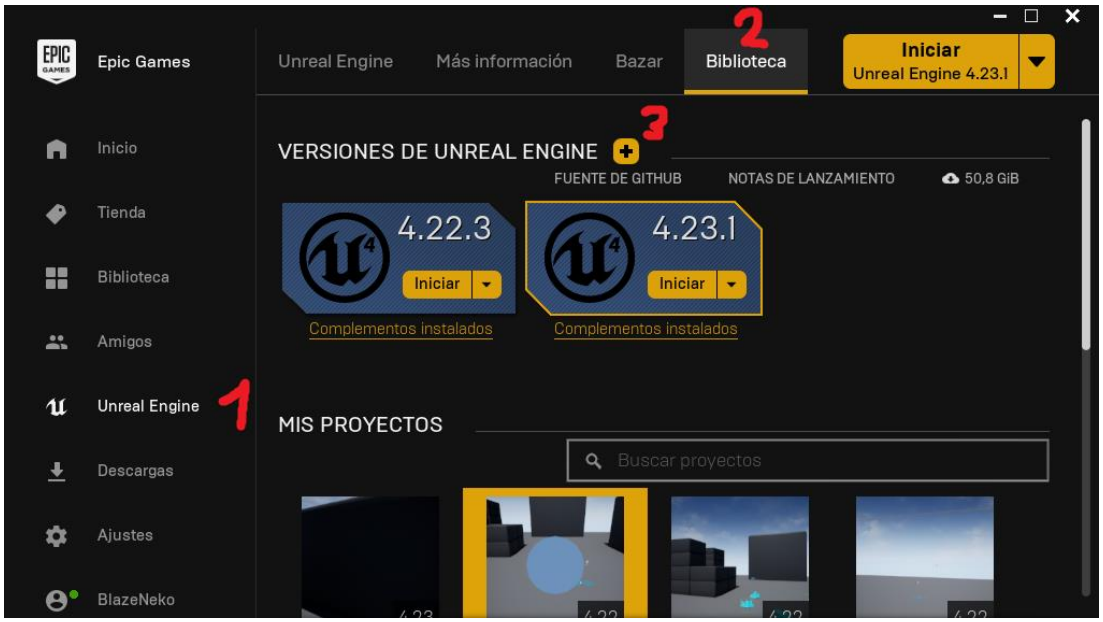

Ilustración 16:Install UE4

Tras iniciar sesión en Epic Games, seguimos los pasos indicados en la imagen anterior.

- Click en Unreal Engine
- Click en Biblioteca
- Click en el "+" para añadir la **versión 4.23** de Unreal Engine

Una vez instalado Unreal Engine, instalamos Visual Studio con el otro instalador de la carpeta.

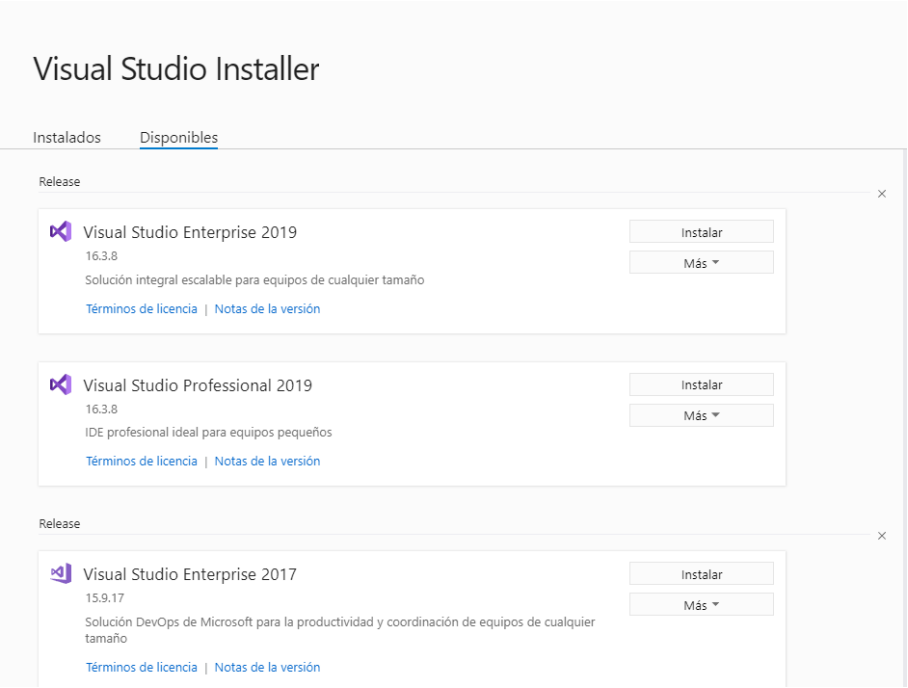

Ilustración 17:Instalar Visual Studio, parte 1

Cuando lleguemos al menú de arriba hay que buscar "Visual Studio Community 2019" y darle a Instalar. Después hay que seleccionar el paquete "Desarrollo de juegos con C++".

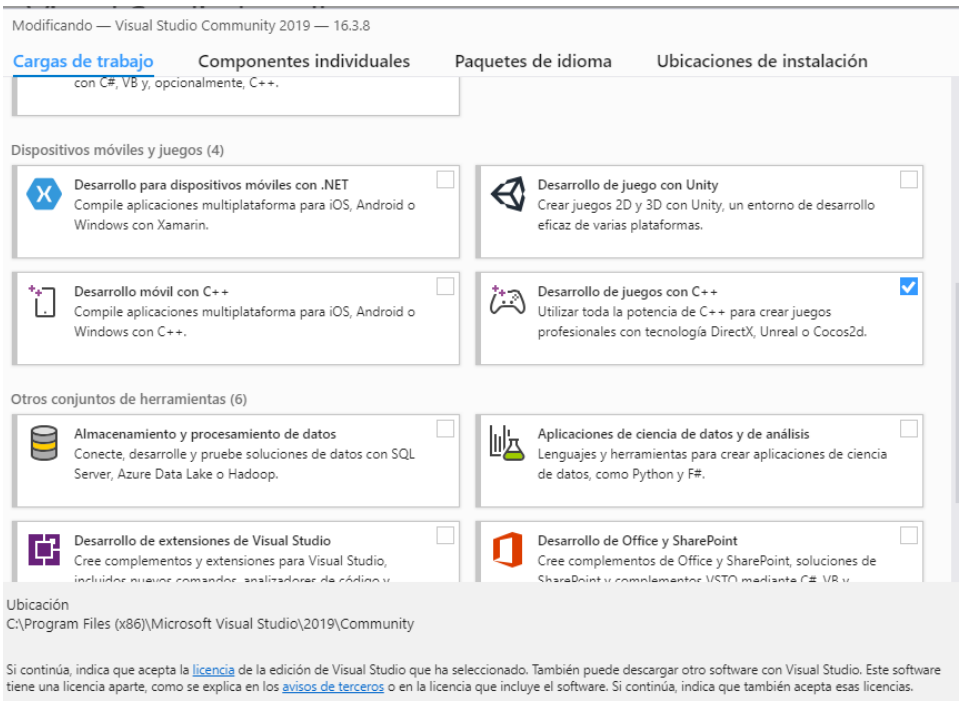

Ilustración 18:Instalar Visual Studio, parte 2.

8.2. Introducción a Unreal Engine 4

Una vez instalados ambos abrimos Unreal Engine y creamos un proyecto vacío para poder acceder a la configuración de Unreal Engine.

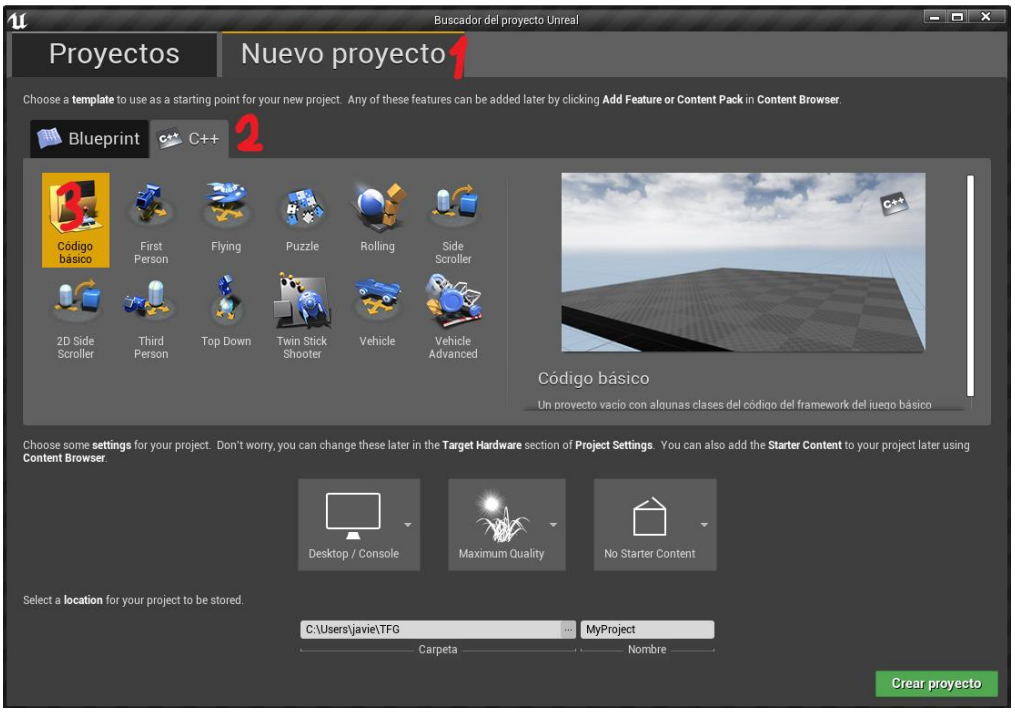

Ilustración 19:Crear Proyecto

Una vez creado el proyecto vamos a los menús desplegables de arriba y seleccionamos **Editar > Preferencias del Editor**.

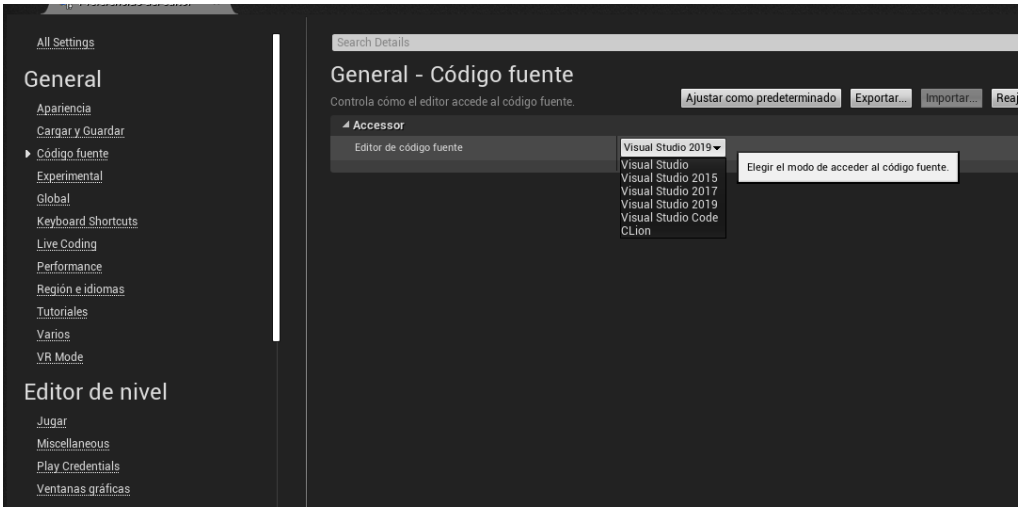

Ilustración 20:Seleccionar IDE

Desde ahí seleccionamos **General > Código Fuente** y ponemos **Visual Studio 2019.**

Ya podemos borrar el proyecto que acabamos de crear y abrir el trabajo de fin de grado. Nos vamos a la carpeta TFG y hacemos **click derecho en TFG.uproject > Generate Visual Studio Project Files.**

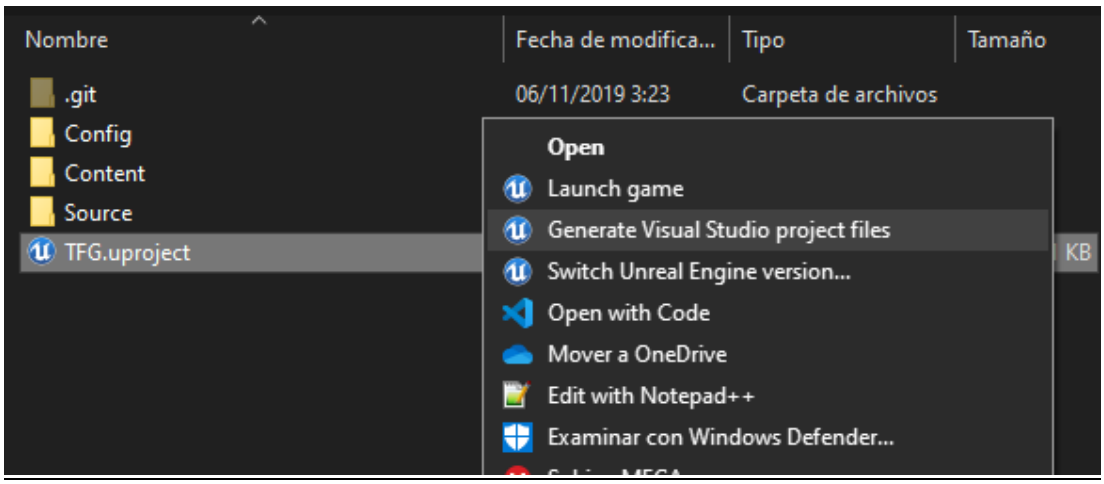

Ilustración 21:Generar ficheros de Visual Studio

Una vez se han generado los ficheros volvemos a abrir Unreal Engine. Esta vez importamos el proyecto haciendo click en Examinar y buscamos el fichero TFG.uproject.

Al abrir el proyecto aparecerá el siguiente mensaje:

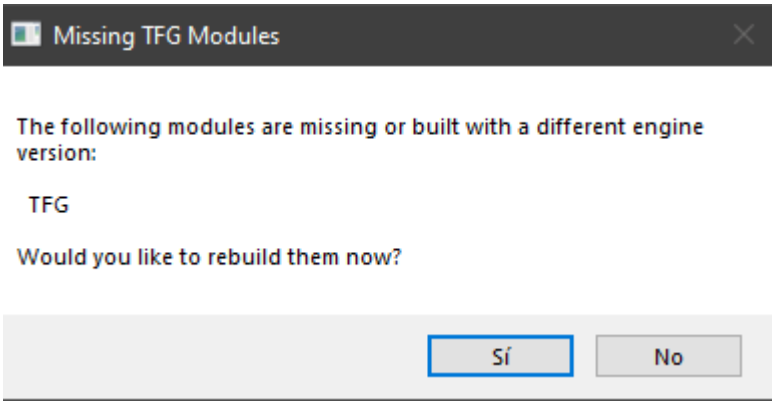

Ilustración 22:Importar proyecto

Le damos a "Sí" y nos cargará el proyecto.

Antes de nada, vamos a darle a **Archivo > Actualizar Visual Studio 2019 Proyecto.**

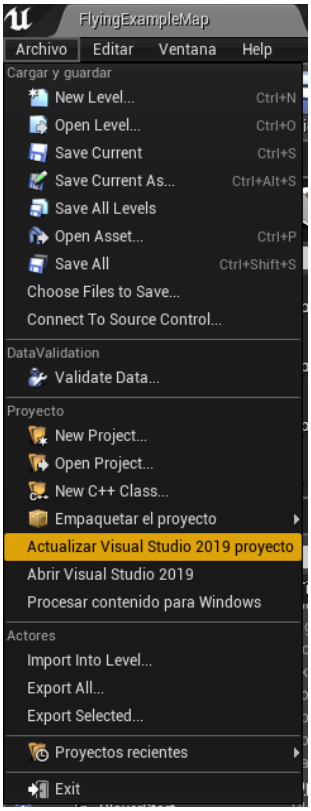

Ilustración 23: Actualizar el poyecto

Con eso termina toda la preparación. Ya podemos probar el proyecto.

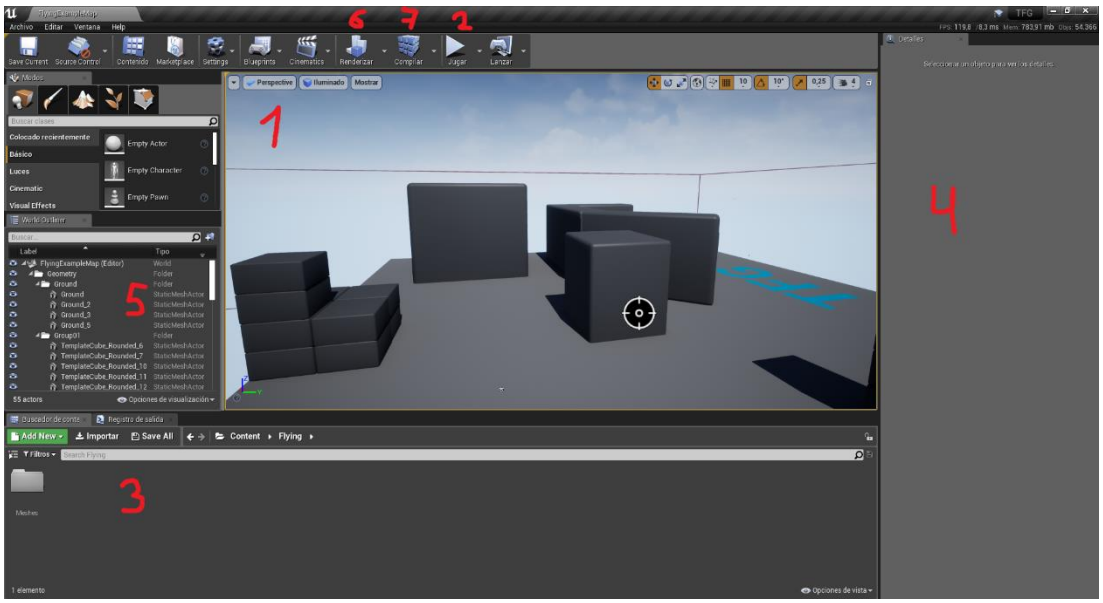

Ilustración 24:El editor de Unreal Engine

1. Viewport en la que vemos el escenario. Para desplazarnos por ella tenemos que mantener pulsado el click derecho del ratón y movernos con W, A, S, D. La rueda del ratón permite aumentar o disminuir la velocidad de movimiento.
2. Botón para ejecutar el proyecto y verlo en funcionamiento
3. Buscador de Contenido. Nos permite movernos por todo el contenido del proyecto.
4. Al hacer click en un objeto de la viewport o del World Outliner (la ventana con el 5), podemos ver información sobre esos objetos en esa columna.
5. El World Outliner es una lista de todos los objetos colocados en el escenario. No todos son visibles en el viewport. La única forma de ver el "Plane Spawner" es a través del World Outliner.
6. Es posible que al importar el proyecto salga un error que diga: "Lighting has to be rebuild". Se soluciona haciendo click en el icono de Renderizado para que renderice de nuevo la iluminación de la escena.
7. Compilar el proyecto.

Los Blueprints se encuentran todos en la carpeta Content > Blueprints.

Algunos de esos blueprints ya no forman parte de la implementación final, pero formaron parte del prototipado y están completamente documentados.

Para ver el código C++ hay que darle a la opción marcada con 1 en la imagen siguiente y marcar "Show C++ Classes". Luego hacemos click en el icono del la carpeta y seleccionamos C++ Classes.

Todas las clases C++ forman parte de la implementación final.

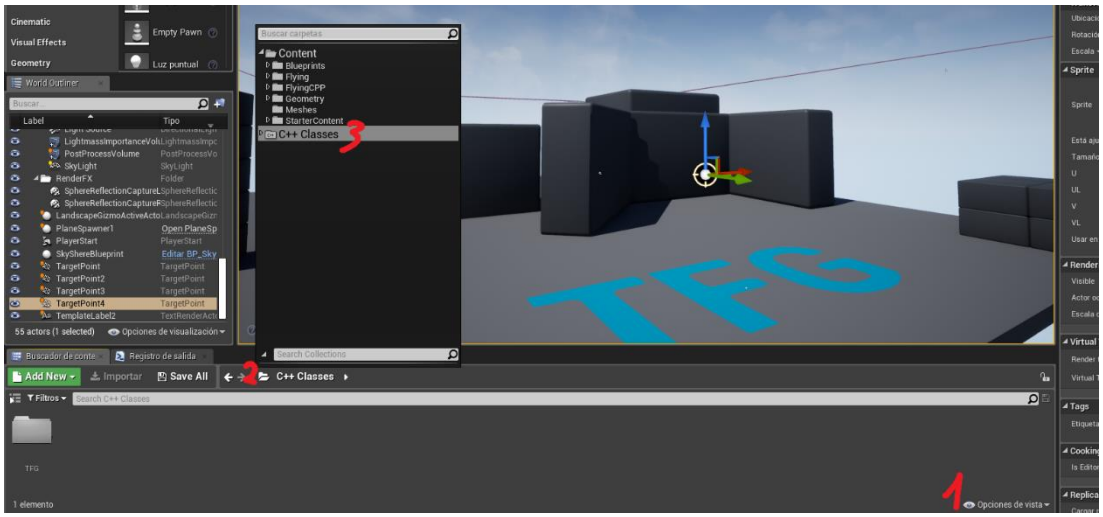

Ilustración 25:Localizar ficheros C++

8.3. El proyecto desde el editor

Si se quieren modificar ciertos parámetros de cara a la ejecución, se puede abrir la blueprint de los aviones. Se encuentra ubicada en **Contents > Blueprints > Plane.**

Haciendo doble click en una Blueprint se abre el editor de Blueprints.

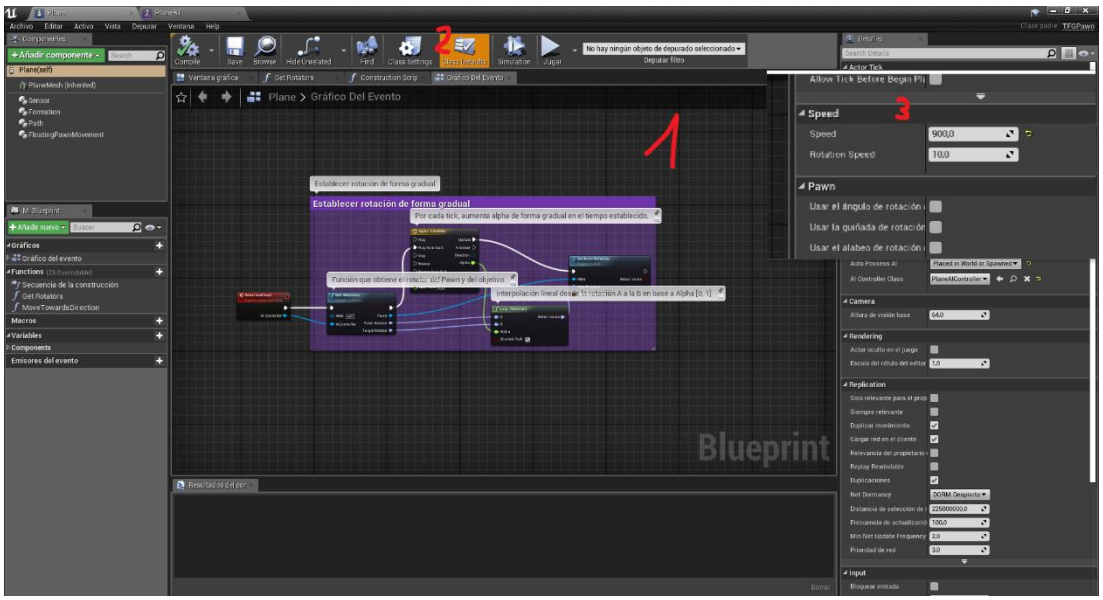

Ilustración 26:Editor de Blueprints

1. Visor de Blueprints

2. Vista de los atributos por defecto de la clase (Se ven en la columna de la derecha). Son modificables sin necesidad de compilar.
3. La zona con Zoom se corresponde con los atributos para modificar la velocidad de los aviones. Son atributos heredados de la clase C++.
4. En la columna de la izquierda podemos seleccionar los componentes de la Blueprint (y visualizar/modificar sus variables expuestas) o asignar componentes nuevos. Todas las instancias de esa blueprint incluirán los componentes con los mismos valores por defecto.

Los árboles de comportamiento se encuentran en la carpeta **Content >Blueprints > BT.** Dentro de esa carpeta se encuentran también las carpetas Services (Blueprints de servicios, explicadas en el apartado 9.8.1, utilizadas en la versión final del proyecto) y Tasks (Blueprints de tareas. Todas forman parte del prototipado y no están implementadas en la versión final del proyecto).

9. MANUAL DEL PROGRAMADOR

El proyecto sigue el paradigma de programación orientada a componentes. Las clases las he diseñado de manera que sean independientes de los componentes, lo que facilita el mantenimiento y la reutilización del código.

En esta sección explicaré el funcionamiento de todo el proyecto. Para información más exhaustiva sobre cada variable y funciones, recomiendo consultar la documentación interna.

Para más información sobre las Blueprints implementadas, recomiendo abrirlas en el editor debido a que no caben en el documento (aunque hablaré de ellas). Están completamente documentadas.

9.1. API de Unreal Engine 4

Toda la API de Unreal Engine gira en torno a Actores y Objetos. Un Actor es toda clase derivada de AActor. Los actores son aquellas clases que pueden ser instanciadas en el mundo que estamos creando.

Los objetos son todas las clases derivadas de UObject. Todas las clases de Unreal Engine derivan de UObject, por lo que los Actores también son objetos, pero se suele referir como Objetos a aquellas clases que NO derivan de AActor.

Según la nomenclatura de Unreal Engine, toda clase derivada directamente de UObject debe llevar una U al principio del nombre, mientras que toda clase derivada de AActor debe comenzar por una A.

9.2. Diagrama conceptual

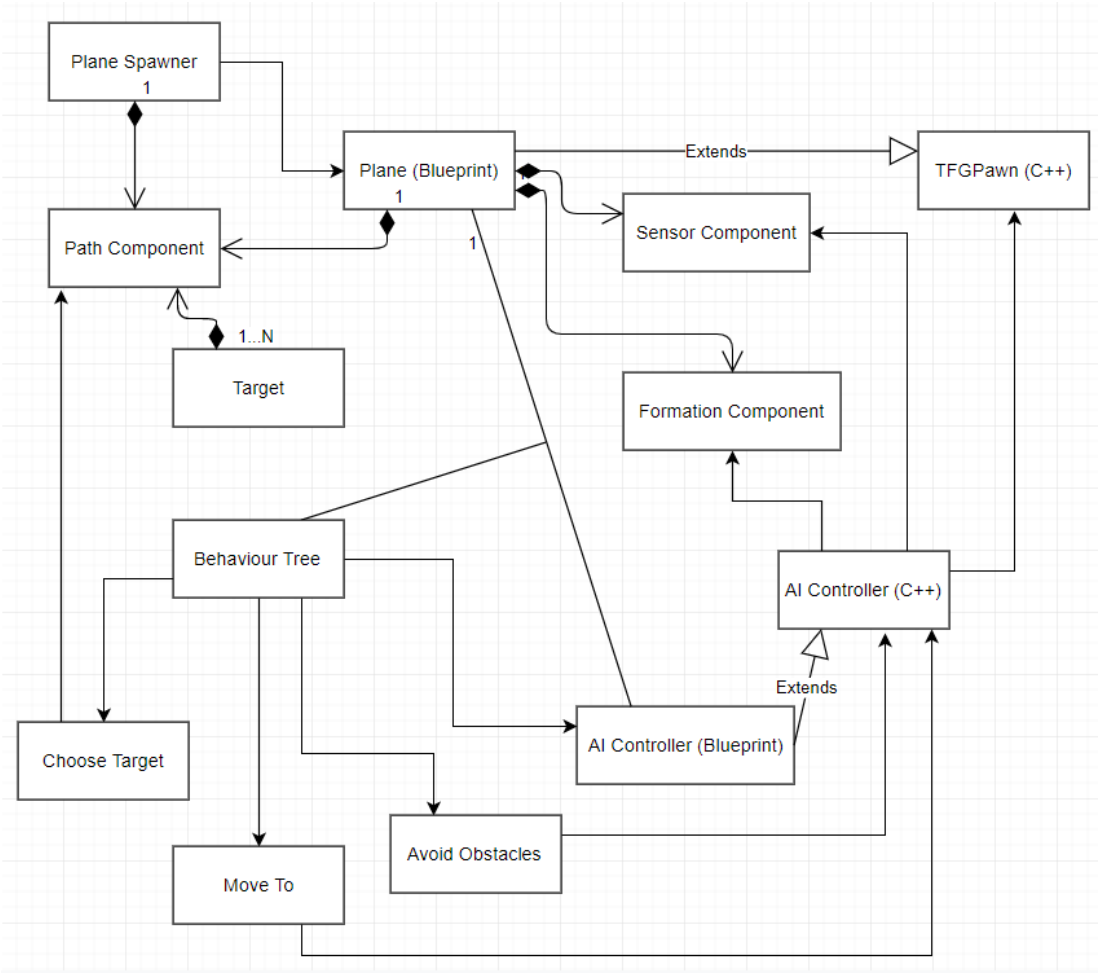

Ilustración 27:Diagrama Conceptual

9.3. Targets

Un ejemplo de Actor son los Targets que he repartido por el mapa o los obstáculos.

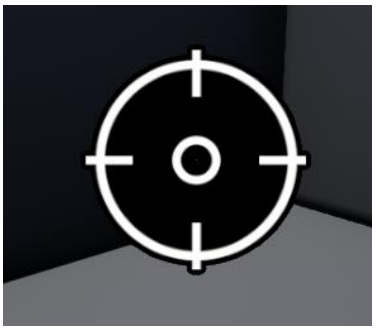

Ilustración 28:Target

Los targets los gestiona un componente llamado **Path Component**.

9.4. Pawns

Los Pawns son actores que pueden ser poseídos por un controlador (ya sea por un Player Controller para recibir un input de un usuario o por un AI Controller para que sean controlados por una AI o algoritmos).

Para este proyecto he creado la clase ATFGPawn con dos métodos que permiten únicamente moverse y girar, pero para ello necesitan recibir un input.

Posteriormente creé una Blueprint llamada "Plane" derivada de los ATFGPawns. El código C++ puede ser accedido desde las Blueprints (tanto funciones como variables) si así es especificado en el código.

Tambien podemos exponer variables para poder ser editadas desde el editor de Unreal. Esta facilidad para utilizar variables en múltiples sitios es extremadamente útil, ya que te permite cambiar los valores al instante sin necesidad de compilar el código.

```
UPROPERTY(EditInstanceOnly, Category = "Path")
TArray<AActor*> pathArray;
```

Ilustración 29:La macro UPROPERTY nos permite, entre otras cosas, visualizar variables en Blueprints o el editor

Todos los aviones que se instancian en el proyecto son Blueprints.

El movimiento depende de que el Pawn tenga un componente de movimiento. Hay varios componentes de movimiento que viene n incluidos en la API de Unreal Engine, concretamente he usado "**UFloatingPawnMovement**", que es para calcular el movimiento de los peones flotantes.

La clase C++ del avión tiene 2 variables expuestas para poder ser editadas en el editor o en blueprints: Speed y Rotation Speed.

Tanto la Speed como la Rotation Speed van multiplicadas por el delta time (tiempo entre frames) para que vaya en todos los ordenadores a la misma velocidad.

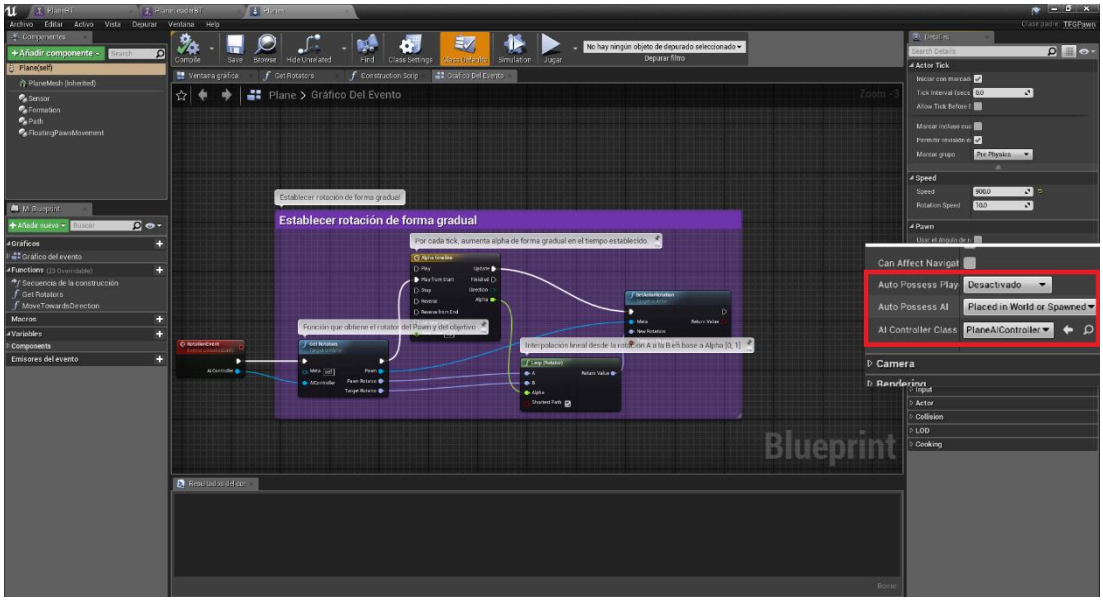

Ilustración 30:Editor de Blueprints

Las Blueprints son extremadamente útiles a la hora de inicializar clases. En la imagen anterior se puede ver el editor de Blueprints con la blueprint de los aviones abierta.

La parte que más interesa son los campos "Auto Posess AI" y "AI Controller Class". Con esos campos se decide si el Pawn es controlado por un jugador o un AI Controller, y podemos especificar qué AI Controller queremos usar y cuándo queremos que lo posea. En este caso los Pawns son poseídos cuando son colocados en el mundo o cuando son generados en él.

Para ello creé un AI Controller en C++ y una Blueprint que deriva de él llamada "**PlaneAIController**".

9.5. AI Controller

El controlador de AI mediante C++ se encarga de calcular la dirección a seguir por los aviones y aplica el input a los aviones.

La blueprint del controlador de AI se encarga de asignar árboles de comportamiento a los aviones. Hablaré más de él cuando llegue a los árboles de comportamiento y las tareas.

9.6. Plane Spawner

Para poder crear formaciones de tamaño variable, decidí crear un generador de aviones.

El Plane Spawner es un actor implementado en C++ encargado de generar los aviones e inicializar su ruta y la formación.

Tiene 3 campos expuestos al editor para inicializar los aviones y un Path Component. Como inicialmente no tenemos ningún avión colocado en el escenario, no se les puede asignar una ruta a no ser que sea en tiempo de ejecución. Para ello inicializamos la ruta a seguir en el Plane Spawner y se la copiamos a los aviones cuando los genera.

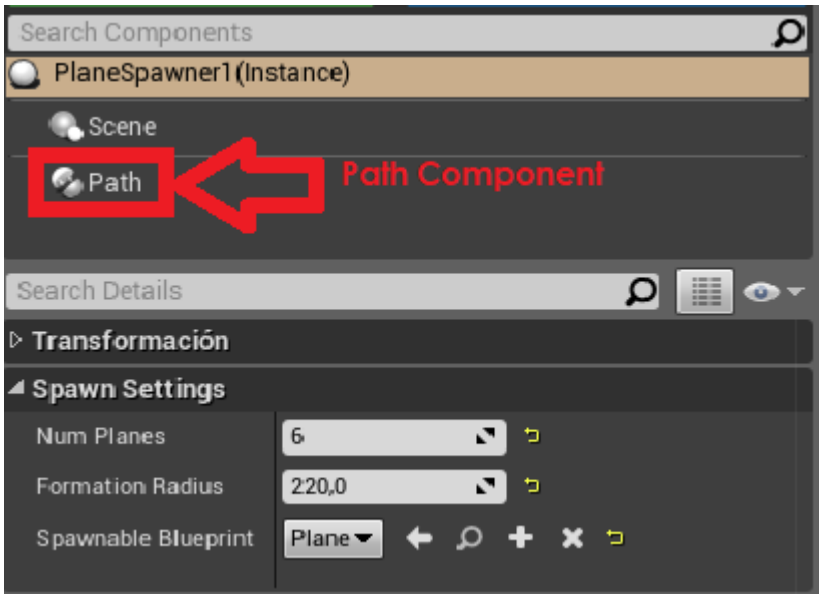

Ilustración 31:Plane Spawner

Las formaciones se generan tal y como aparecen especificadas en el apartado 5.5 de este documento.

Num planes indica el número de aviones que queremos generar, formation radius indica el radio de la formación y "Spawnable Blueprint" es la Blueprint que queremos generar (aviones).

9.7. Componentes

Todos los componentes van unidos a los aviones. El Path Component además va acoplado al Plane Spawner.

9.7.1. Path Component

Simplemente un TArray (Arrays de Unreal Engine) que almacena los Targets. Sus únicos métodos son un Get y un Set.

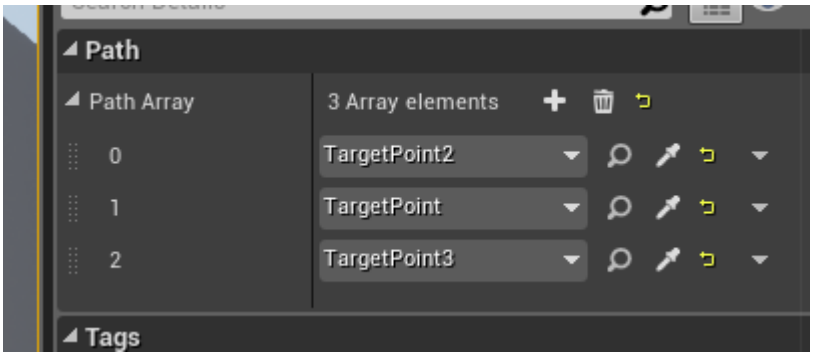

Ilustración 32:Visualización de las variables de la imagen anterior en el editor, con valores ya asignados

9.7.2. Sensor Component

El Sensor Component consta de un sensor frontal que lanza un rayo en forma de caja para detectar obstáculos. Si se ha detectado, emite rayos a lo largo de 2 sensores de 180º hasta que el avión esquiva el obstáculo. Entre sus atributos encontramos:

- **Resultados de cada rayo** atributos públicos que indican principalmente los resultados de cada rayo e información sobre la ubicación del obstáculo más cercano.
- **Parámetros de configuración** como el alcance de los rayos, el número de rayos o la distancia máxima a la que detectar obstáculos. Es modificable desde el editor.
- **Modo debug** que permite visualizar los rayos.

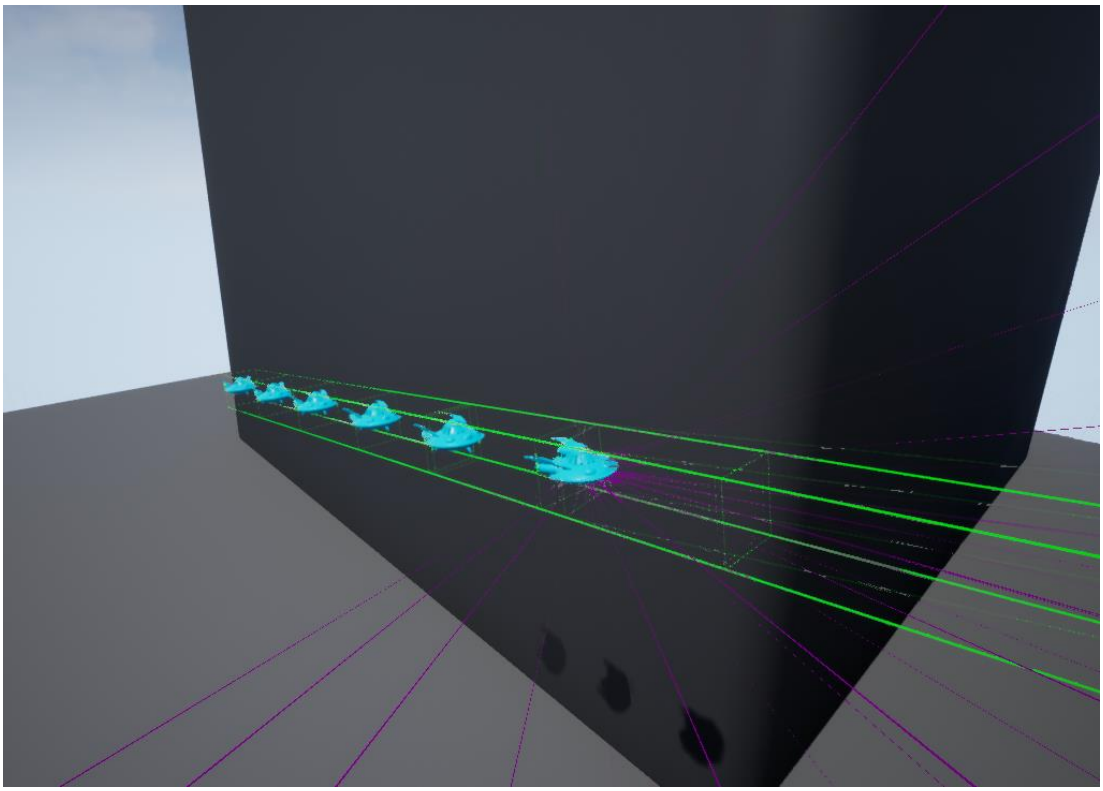

Ilustración 33:Sensor Component

El Sensor Component dispone de 4 funciones principales:

- **LaserSetupSlerp**: Almacena en un array toda la información sobre la dirección a la que se emiten los láser. Hace uso de la librería FMath de la API de Unreal. Este método fue mi primer intento de implementar los lásers y se basa en el cálculo de interpolaciones esféricas entre Quaternions. Actualmente no uso este método porque al rotar los aviones los cálculos de la interpolación esférica eran erróneos pero contiene toda la base que utilicé para crear el siguiente:
- **LaserSetupLerp**: Unreal trata todas las rotaciones internamente como Quaternions, pero tiene una abstracción más sencilla de usar llamada FRotator. (descompone un Quaternion en rotación sobre los ejes X, Y y Z).

  Este método hace lo mismo que el anterior, pero utilizando FRotators en vez de Quaternions, e interpolaciones lineales en lugar de interpolaciones esféricas. Para lograr que las

interpolaciones lineales den un resultado curvo en vez de recto, realizo la siguiente operación sobre cada cada rayo:

"Longitud = longitud_lerp * radio/(longitud_lerp * radio)".

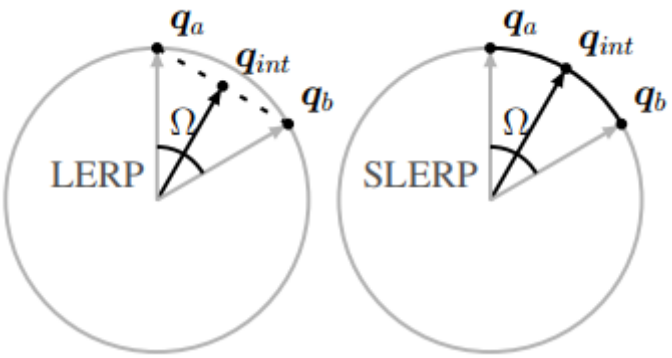

Ilustración 34:Interpolaciones lineales y esféricas

- **rayCast:** Dispara los láser. Inicialmente lanza ÚNICAMENTE el rayo con forma de caja. Si se detecta un obstáculo, lanza todos los rayos calculados previamente por **LaserSetupLerp**.
- **debugLines:** Método que dibuja las líneas de debug. Siempre muestra todos los sensores aunque realmente no se estén lanzando rayos mediante los sensores horizontal y vertical, pero es lo que necesitaba para el debug.
- **checkRayResult:** Calcula a partir de los resultados obtenidos por rayCast la dirección de donde proviene el obstáculo (si se ha detectado alguno). También establece el valor "true" o "false" en la variable "avoid", que se utiliza en el árbol de comportamiento.

9.7.3. Formation Component

Almacena todos los datos relativos a la formación. Este componente es inicializado por el Plane Spawner.

Posee dos TArrays: Uno que almacena todos los pawns de la formación y otro que contiene los offsets respecto al líder.

9.8. Behaviour Trees

Los Behaviour Trees son árboles que permiten ejecutar tareas según una lógica almacenada en una pizarra. Los behaviour trees se componen de 3 elementos: Tareas, Decoradores y Servicios.

Los behaviour trees parten siempre de un nodo "Root". Los nodos deben ir enlazados siempre a una Task o a un Composite Node.

¿Qué es un Composite Node?

Hay únicamente 3 tipos de Composite Nodes:

- **Sequence**: Ejecuta todos los nodos conectados a la secuencia de izquierda a derecha hasta que uno falla.
- **Selector**: Ejecuta todos los nodos conectados al selector de izquierda a derecha hasta que uno acierta.
- **Simple Parallel**: Ejecuta hasta 2 nodos a la vez.

Los composite nodes son nodos grises en los árboles de comportamiento.

¿Qué son los decoradores?

Añaden condiciones (normalmente basadas en las variables de la pizarra) y bucles a Composite Nodes y a Tareas. Se pueden configurar para que el nodo devuelva "fallo" cuando se deja de cumplir la condición, así se puede forzar a que continúe un selector o se interrumpa una secuencia. Los decoradores son bloques azules en los árboles de comportamiento.

¿Qué son los servicios?

Los servicios se unen a Composite Nodes o a tareas y se ejecutan en la frecuencia indicada siempre y cuando la rama del árbol a la que pertenecen se encuentra activa. Se suelen utilizar para actualizar valores de la pizarra. Los servicios son bloques verdes en los árboles de comportamiento.

¿Qué son las tareas?

Las tareas son funciones programadas en C++ o Blueprints que son ejecutadas cuando el árbol selecciona el nodo de la tarea a ejecutar. Son los nodos morados de los árboles de comportamiento.

Blackboard

La Blackboard o pizarra es una estructura que almacena variables de un Behaviour Tree. Una pizarra puede ser utilizada por varios árboles distintos. Soporta muchos tipos de variables (como objetos o actores) y las variables se pueden establecer como únicas de cada instancia o compartidas por todas las instancias de esa misma pizarra.

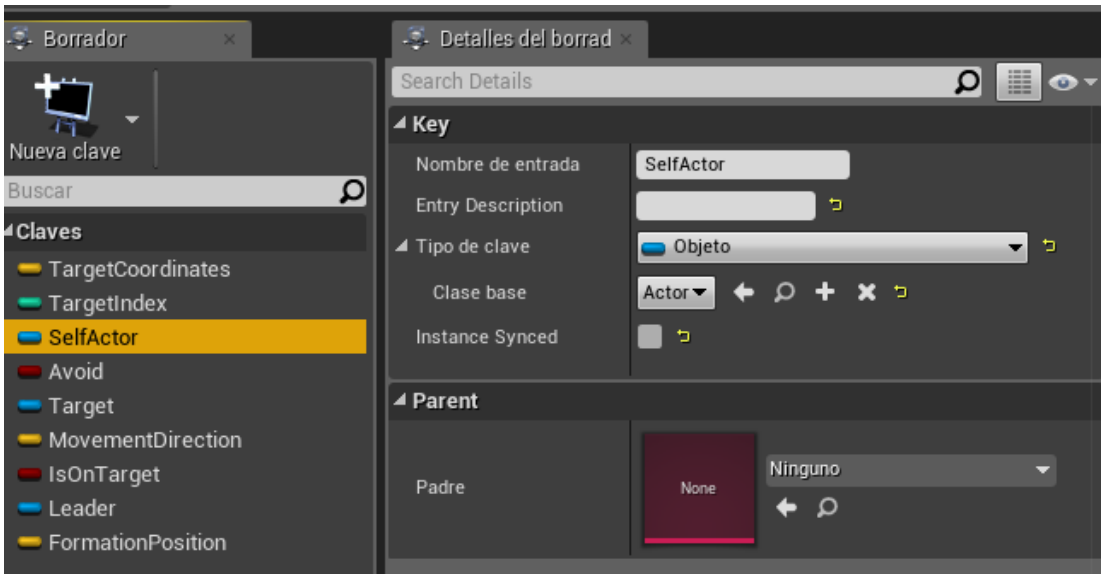

Ilustración 35:Pizarra

Cada instancia del **Plane AI Controller** utiliza su propio árbol y su propia pizarra. Esto lo hago mediante Blueprints y, además, comprueba si el peón al que posee es el líder de la formación o no para utilizar el árbol correspondiente

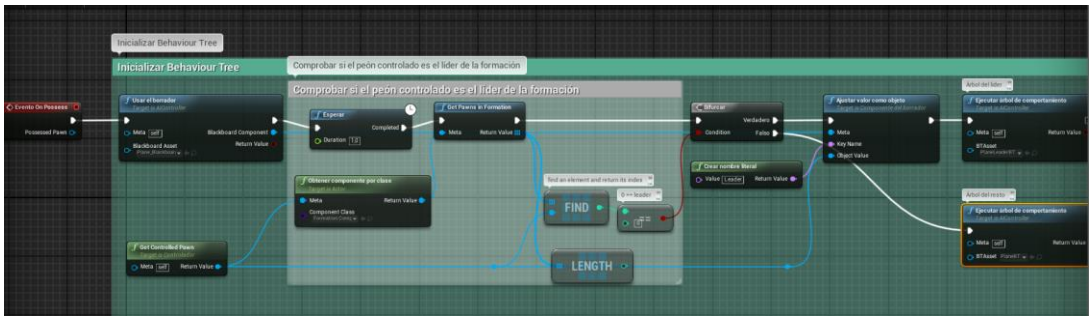

Ilustración 36:Blueprint del AI Controller

Las tareas implementadas están explicadas en el apartado 5.6.

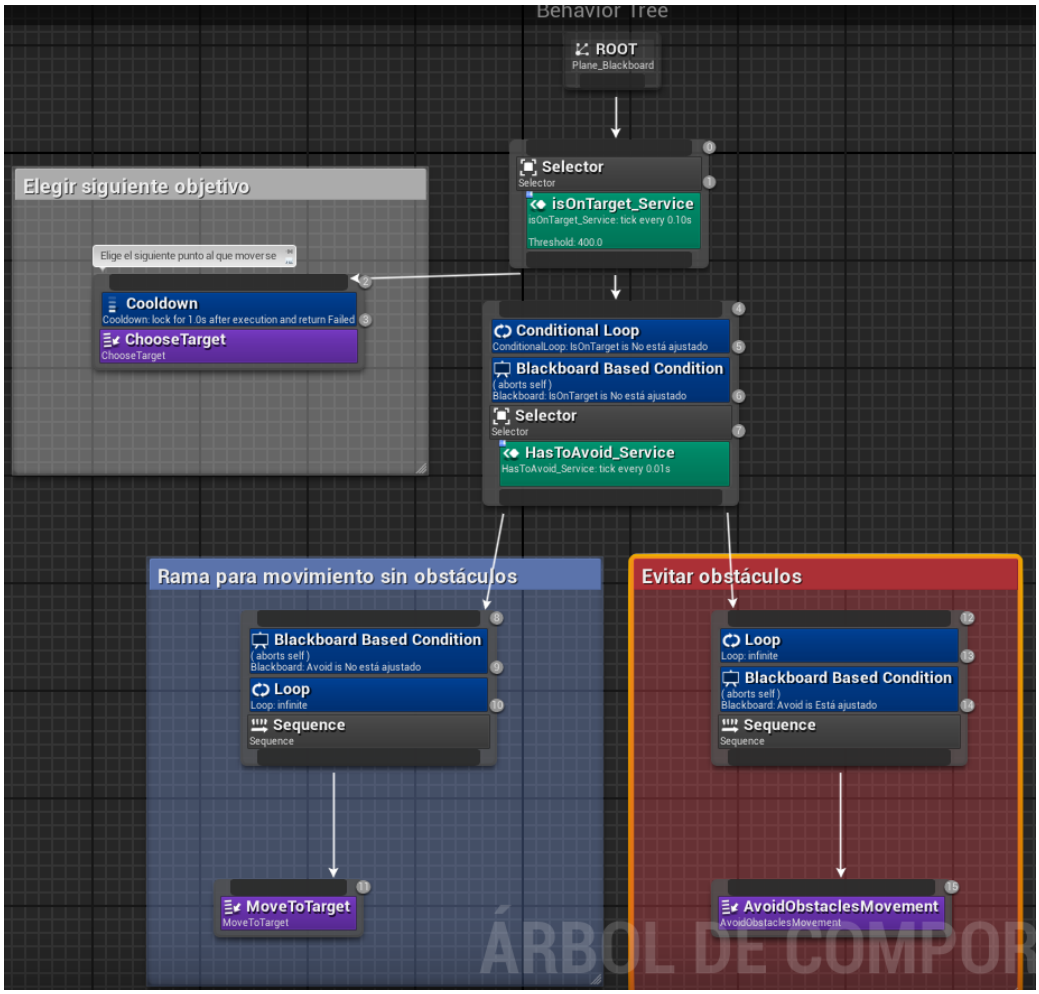

Ilustración 37: Behaviour Tree del líder

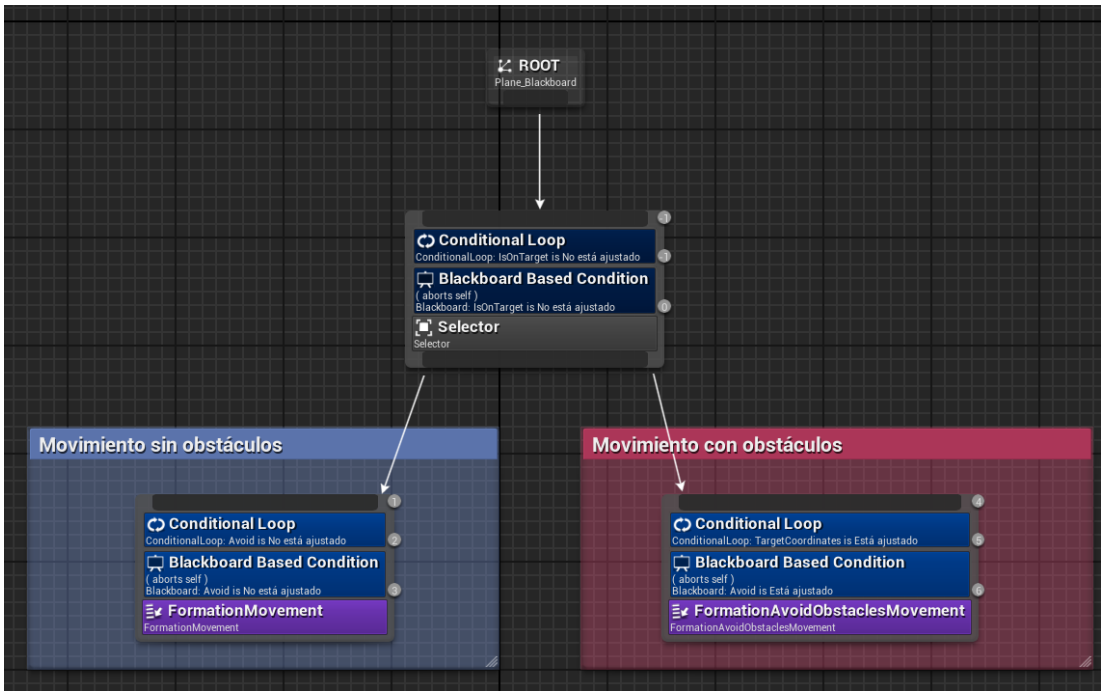

Ilustración 38:Behaviour Tree del resto de aviones

9.8.1. Servicios

He implementado dos servicios, ambos mediante blueprints debido a su simplicidad.

Uno de ellos comprueba si el avión ha llegado al target y actualiza la variable isOnTarget de la Blackboard. El otro servicio comprueba si se ha detectado un obstáculo (copiar y pegar el valor de la variable "avoid" del sensor component en la pizarra).

Para más información sobre las Blueprints implementadas, recomiendo abrirlas en el editor debido a que no caben en el documento (aunque hablaré de ellas). Están completamente documentadas.

10. CONCLUSIONES Y LÍNEAS FUTURAS

Este proyecto me ha servido mucho para aprender Unreal Engine 4. Esto me es extremadamente útil de cara al máster que estoy cursando y al futuro laboral, pues me quiero dedicar a la industria de los videojuegos.

La complejidad ha estado sobre todo en aprender a manejar el motor y ponerme al día con las matemáticas necesarias para llevar a cabo el proyecto.

Al principio opté por utilizar una implementación ya existente de un algoritmo A* y trabajar más a fondo en las formaciones, pero vi más satisfactorio realizar mi propio algoritmo de búsqueda a ciegas, pues apenas hay implementaciones de ellos.

Hay cosas que se podrían implementar en un futuro para expandir el programa:

1) Interfaz de usuario configurable

Una interfaz de usuario que permita configurar con facilidad las variables deseadas sin recurrir al editor de Unreal Engine.

2) Más formaciones

El proyecto sólo tiene de momento la formación circular y la formación en línea recta. Se le podrían añadir más formaciones (En flecha, en cono, en cuadrado…)

3) Cambios de líder

El líder podría cambiar cada vez que se alcanza un obstáculo (o no, a elección del usuario)

4) Mejorar el algoritmo para que funcione de forma más precisa cuando los aviones se encuentran atrapados**.**

5) Obstáculos móviles

BIBLIOGRAFÍA

Encontrar información para manejarme en Unreal Engine fue lo más complicado. La API es inmensa y no es fácil introducirse a ella. Casi toda la información que se puede encontrar en Google está en Blueprints y no en C++, por lo que tuve que apoyarme mucho en un curso de Unreal Engine.

- **Documentación de Unreal Engine 4**

  https://docs.unrealengine.com/en-US/index.html

  (Fecha de último acceso: 4 de Septiembre de 2019)

- **Foros de Unreal Engine 4**

  https://forums.unrealengine.com/

  Los foros de Unreal Engine 4 han sido como el Stack Overflow de Unreal Engine. Muchas dudas han sido resueltas con realizar algunas consultas allí.

  (Fecha de último acceso: 1 de Septiembre de 2019)

- **IA voladora en Unreal Engine 4 con Blueprints**

  https://www.youtube.com/watch?v=G5bJP5Nr88o

- **DoN's 3D Pathfinding** (No está tan relacionado ya que es un algoritmo A*, pero es el video que me dio la idea para el proyecto)

  https://www.youtube.com/watch?v=6Tr_K551zvI

- **Cursos de Udemy:** Mi mayor ayuda y lo que realmente me ha ayudado a sacar adelante el proyecto. Los dos enlaces son del mismo curso, pero el segundo es la versión antigua (no está actualizada para las nuevas versiones de Unreal). El curso antiguo tiene un proyecto adicional que trata sobre IA y árboles de comportamiento.

  https://www.udemy.com/course/unrealcourse/

  https://www.udemy.com/course/unreal-engine-c-developer-archived-course/

(Fecha de último acceso: 2 de Septiembre de 2019)

- Información sobre formaciones de objetos en los foros de Unity, en C#

  https://forum.unity.com/threads/rts-unit-formation.169319/

  (Fecha de último acceso: 2 de Septiembre de 2019)

- Más información sobre formaciones, aunque en un entorno 2D. Pude sacar algunas ideas de ahí.

  https://www.gamasutra.com/blogs/DruErridge/20180522/318413/Group_Pathfinding__Movement_in_RTS_Style_Games.php

  (Fecha de último acceso: 4 de Septiembre de 2019)

ANEXOS

En la carpeta "Blueprint Images" he añadido imágenes de todas las Blueprints. Todas ellas tienen bloques de comentarios.

- "AIController-InitializeBehaviourTrees.PNG" es la Blueprint del controlador de AI. Se encarga de inicializar los árboles de decisiones de cada avión en el momento que los poseen. Utilizada en la implementación final.
- "Plane-GetRotators.PNG" es una función auxiliar que me permite obtener la rotación actual de un avión y la rotación necesaria para mirar al objetivo. La función devuelve ambas rotaciones. Empleada únicamente en el prototipo.
- "Plane-Rotation.PNG" es la función encargada de rotar el avión hacia el objetivo. Se vale de la función mencionada en el párrafo anterior. Empleada únicamente en el prototipo.
- "TreeService-HasToAvoid.PNG" Se trata de un servicio para los behaviour trees que comprueba si hay que evitar algún obstáculo para actualizar una variable en la pizarra. Se utiliza tanto en el prototipo como en la implementación final
- "TreeService-IsOnTarget.PNG" Se trata de un servicio que comprueba si el avión se encuentra en el objetivo para actualizar una variable en la pizarra. Se utiliza tanto en el prototipo como en la implementación final.
- "TreeTask-GetAvoidanceDirection.PNG" Es una tarea que calcula la dirección que hay que seguir para evitar un obstáculo ya detectado y la guarda en la pizarra. Se utiliza únicamente en el prototipo.
- "TreeTask-GetDirectionToTarget.PNG" Es una tarea que calcula la dirección hacia el objetivo y la guarda en la pizarra. Se utiliza únicamente en el prototipo.
- "TreeTask-MoveToDirection.PNG" Es una tarea que mueve el avión hacia la dirección indicada en la pizarra (que será la necesaria para evitar el obstáculo o la dirección hacia el objetivo). Se utiliza únicamente en el prototipo.

- "TreeTask-RotateToTarget.PNG" Es una tarea que rota el avión hacia el objetivo. Se utiliza únicamente en el prototipo.
- "TreeTask-RotateTowardsDirection.PNG" Es una tarea que rota el avión hacia la dirección indicada en la pizarra. Se utiliza únicamente en el prototipo.

Todas las blueprints que se utilizan únicamente en el prototipo las implementé posteriormente en C++. Las tareas de C++ se componen de una o más tareas que fueron implementadas en Blueprints.