



Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**



Grado en Ingeniería Informática

Trabajo Fin de Grado

**Diseño y Desarrollo de un Prototipo de
Simulación para Robots Aéreos basado
en Unreal 5, ROS2 y Gazebo (Informe
Intermedio)**

Autor: Daniel Corrales Falco
Tutor(a): Santiago Tapia Fernandez

Madrid, Abril 2023

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

*Trabajo Fin de Grado
Grado en Ingeniería Informática*

Título: Diseño y Desarrollo de un Prototipo de Simulación para Robots Aéreos basado en Unreal 5, ROS2 y Gazebo

Madrid, Abril 2023

Autor: Daniel Corrales Falco

Tutor: Santiago Tapia Fernandez

Dpto. Lenguajes de Sistemas Informáticos e Ingeniería del Software
ETSI Informáticos

Universidad Politécnica de Madrid

Resumen

Los drones son vehículos aéreos no tripulados que se crearon en un inicio con fines militares, sin embargo, hemos sido capaces de adaptarlos a una amplia variedad de funciones tanto en el ámbito militar como civil.

La simulación de drones es un campo relativamente nuevo que, sin embargo, ha experimentado un desarrollo exponencial en los últimos años. Durante este tiempo se han realizado significativos avances en el estudio de la física del movimiento de los drones, un aspecto esencial para su posterior simulación en entornos virtuales, que permiten explorar las posibles aplicaciones de estas versátiles máquinas.

Como objetivo principal este trabajo pretende establecer una base para la creación de un simulador de vuelo de drones en el entorno gráfico de Unreal Engine 5 haciendo uso de la librería de comunicación de robots popularmente conocida como ROS2. Así pues, se ha desarrollado un visualizador de vuelo, un concepto similar al simulador, solo que en este caso, el dron realizará un recorrido preestablecido y mostrará el movimiento en un entorno tridimensional. Otro de los objetivos marcados en este trabajo es dividir en dos módulos principales los objetos más importantes dentro del simulador: el dron y el escenario.

Para cumplir este segundo objetivo se ha realizado un desarrollo dividido en hitos siguiendo una metodología de desarrollo ágil informal. En total se podrán encontrar 7 hitos, 6 de ellos completos y un último que se encuentra en desarrollo.

En estos primeros 6 hitos se realizan progresos en vista a un desarrollo iterativo de este proyecto. En el primero se exploran formas de mover objetos dentro de Unreal, en el segundo se estudia cómo es el proceso de importación de mallas, en el tercero se implementa un control diferencial de la velocidad, en el cuarto se desarrolla un sistema publicador-suscriptor de ROS2, en el quinto se desarrolla un peón dentro de Unreal que haga uso de ese sistema publicador-suscriptor y finalmente en el sexto se documenta la importación de un escenario completo y sus texturas dentro del editor de Unreal.

Todo el código fuente aquí descrito y explicado se puede encontrar en la página de [github](#)[Tap23a] además de recogerse en los anexos de la propia memoria.

Agradecimientos

Quiero expresar mi profunda gratitud a todas las personas que me apoyaron en la realización de este trabajo. Sin su ayuda, este proyecto no habría sido posible.

Agradezco sinceramente a mi director de TFG, el profesor Santiago Tapia, por su orientación y paciencia. También agradezco a mis compañeros de clase por su apoyo constante sin ellos llegar hasta aquí no habría sido posible.

Mi agradecimiento especial va para mis familiares por su apoyo incondicional. En especial a mi padre por siempre ofrecerme su ayuda y a mi madre y hermana por apoyarme siempre.

¡Gracias a todos!

Índice general

1. Introducción	1
1.1. Antecedentes	1
1.2. Objetivos	1
2. Herramientas y estado del arte	3
2.1. Herramientas	3
2.1.1. Unreal Engine 4.27 y 5:	3
2.1.2. Gitkraken:	5
2.1.3. Github:	5
2.1.4. ROS2:	5
2.1.5. Aerostack2:	6
2.2. Estado del arte de los simuladores de drones	6
3. Desarrollo	9
3.1. Descripción General	9
3.2. Hito 1: Exploración del movimiento de un objeto en Unreal	9
3.2.1. Descripción:	9
3.2.2. Desarrollo:	9
3.2.3. Resultados:	14
3.3. Hito 2: Importación de mallas	15
3.3.1. Descripción:	15
3.3.2. Desarrollo:	15
3.3.3. Resultados:	16
3.4. Hito 3: Implementación de un control diferencial de la velocidad	17
3.4.1. Descripción:	17
3.4.2. Desarrollo:	17
3.4.3. Resultados:	20
3.5. Hito 4: Desarrollo de un sistema pub/sub de ROS2	22
3.5.1. Descripción:	22
3.5.2. Desarrollo:	22
3.5.3. Resultados:	24
3.6. Hito 5: Desarrollo del peón en UE5	24
3.6.1. Descripción:	24
3.6.2. Desarrollo:	24
3.6.3. Resultados:	28
3.7. Hito 6: Importación de escenarios a UE5	29
3.7.1. Descripción:	29
3.7.2. Desarrollo:	29
3.7.3. Resultados:	32

3.8. Hito 7: Integración con Aerostack2 (en desarrollo)	32
3.8.1. Descripción:	32
4. Análisis de impacto	33
5. Conclusiones	35
A. Anexos	37
A.1. Código Hito 1:	37
A.2. Código Hito 3:	41
A.3. Código Hito 4:	44
A.4. Código Hito 5:	48
Bibliografía	55

1. Introducción

1.1. Antecedentes

La simulación es un recurso que está adquiriendo una creciente importancia en el marco tecnológico actual, por lo que está experimentando un crecimiento en los últimos años.

Paralelamente, el desarrollo de videojuegos es otro de los ámbitos de mayor auge en los últimos años, que ha requerido importantes mejoras en el software y hardware. Estos avances se han visto impulsados por la competencia entre las grandes empresas de desarrollo tecnológico.

Mi interés en estos campos se basa en dos puntos, por una parte los simuladores representan un recurso con amplias aplicaciones y oportunidades tecnológicas. Y por otra, los videojuegos siempre han sido un aspecto de interés para mí tanto por su parte lúdica como por su complejidad a nivel de programación. En mi opinión el trabajo escogido combina los dos temas, los simuladores y los videojuegos y me proporciona la oportunidad de aprender cómo funciona un entorno de desarrollo de videojuegos actual como es Unreal Engine 5 y cómo es el proceso de desarrollo detrás de un simulador, además de ampliar mis conocimientos en programación con C++.

1.2. Objetivos

Los objetivos propuestos para este trabajo son los siguientes:

1. Analizar los requisitos y especificaciones de un simulador.
2. Diseñar la arquitectura software del mismo.
3. Especificar y documentar la API del simulador en ROS2.
4. Implementar el simulador como un nodo de ROS2 de acuerdo a la API y usando Unreal 5.
5. Especificar y desarrollar la API del simulador con Gazebo.

2. Herramientas y estado del arte

2.1. Herramientas

Las herramientas utilizadas para el desarrollo de este trabajo han sido: Unreal Engine 4.27 y 5, Gitkraken, GitHub, Teams, ROS2 y Aerostack2. Estas serán descritas a continuación, tanto para describir su función y utilidad como para explicar lo necesario para entender los contenidos recogidos en esta memoria.

2.1.1. Unreal Engine 4.27 y 5:

Como se ha mencionado previamente, el simulador se va a desarrollar para el motor de juego Unreal Engine 5 (UE5) de la compañía Epic Games. Esta es una herramienta de última generación que cuenta con los últimos avances gráficos, pudiendo llegar a generar entornos casi idénticos a la realidad, simulando a la perfección grabaciones del entorno natural y no generadas por ordenador. Asimismo, cuenta con un sistema de físicas y colisiones integrado, lo que facilitará las tareas de implementación más adelante. Como añadido, este entorno también cuenta con herramientas de Inteligencia Artificial, que se pueden usar para controlar los actores, refiriéndonos a los drones, que podemos llegar a tener en la escena.

El principal motivo para realizar el desarrollo en este motor, es su gran proyección de futuro, gracias a la capacidad gráfica que proporciona y la oportunidad de hacer uso de IA para entrenar drones in silico para luego poder aplicarlo en máquinas reales.

El motivo principal por el que se han utilizado dos versiones distintas de este software ha sido la limitación técnica de los equipos. El trabajo se ha repartido entre 2 máquinas distintas con capacidades dispares. Sin embargo, esto no supone un problema ya que la única diferencia entre ambas versiones se centra en el apartado gráfico siendo la infraestructura idéntica en ambas. Es decir, el código desarrollado en la versión 4.27 es válido en la versión 5.

Habiendo justificado esto, en vista de que los contenidos del trabajo son técnicos, es necesario explicar algunos conceptos de Unreal Engine.

1. En el editor de Unreal Engine, no importa cual de las dos versiones esté usando, podremos encontrar varios elementos. El primero y más llamativo una ventana la cual nos muestra el nivel en el que se está trabajando actualmente. En este podremos ver las mallas y elementos que el usuario vaya añadiendo. Al mismo tiempo veremos varias pestañas que nos muestran mucha información, de las cuales nos interesan la pestaña de detalles

del mundo y el cajon de contenido o “Content Browser”. En la primera podremos ver en detalle la información que viene asociada a cada elemento presente en el nivel y en la segunda podremos encontrar todos los elementos de los que disponemos en nuestro proyecto. Este cajon de contenido es el sistema de ficheros del proyecto y podremos añadir y eliminar tantos objetos como queramos. Aquí también se crearan las clases C++ de los diferentes objetos, mallas que importemos al proyecto y los blueprints que creamos.

2. Unreal ofrece una amplia variedad de objetos y clases creados por sus desarrolladores para facilitar las tareas de desarrollo de otras personas que deseen utilizar este motor de juego. Las más importantes a las que se da uso en este trabajo son: componentes de escena, actor y peón.
3. La clase actor predefinida por Unreal permite crear un objeto dentro del juego. Este es el objeto más básico al cual se le pueden programar los comportamientos que se deseen, sin embargo, es al mismo tiempo el más limitado dado que no tiene acceso a utilidades más avanzadas de Unreal como entradas de teclado, simulación de físicas o manejo por inteligencia artificial. A pesar de ser tan limitado esta es la clase más importante al ser la clase padre de la que heredarán los objetos más complicados, peón y personaje (“character”). Esta cuenta con las funciones BeginPlay(), EndPlay() y Tick() que son las funciones básicas para hacer funcionar a un objeto de este tipo. La función BeginPlay() es una función que se ejecuta una única vez al comenzar el nivel, esta resulta muy útil para realizar configuraciones iniciales que requiera el actor, como inicializar variables. La función EndPlay() es otra función que se ejecuta una única vez, solo que esta se ejecuta al terminar el nivel, esta normalmente solo imprime mensajes en la consola para indicar que el actor se ha retirado de la escena o nivel. Estas dos funciones permiten ser sobreescritas (“overridable”) por lo que se pueden hacer a medida de la necesidad de cada desarrollador tanto en la clase actor como en las que hereden de la misma. La función de Tick() es una función que se comporta como un bucle, ejecutándose a cada fotograma generado por Unreal durante la ejecución del nivel.
4. Los componentes de escena son componentes que podemos usar como apoyo para incluir funcionalidades a los actores. Además podemos encontrar otro tipo de componentes, los componentes de actor, estos no hacen nada esencialmente, sin embargo es imprescindible dado que es la base de la jerarquía del resto de componentes.
5. Finalmente tenemos la clase peón, esta como se ha mencionado previamente hereda de la clase actor, teniendo acceso a las 3 funciones principales que esta contiene. Sin embargo, un peón es más avanzado que un actor, pues estos tienen acceso al control por inteligencia artificial y a las entradas de teclado. En cuanto a las físicas que soporta, estas son las básicas ofrecidas por el editor sin ser las físicas avanzadas que pertenecen a la clase de personaje, la cual se centra en simular a un ser humano. A pesar de esto, esta clase es la que mejor encaja con los objetivos del trabajo puesto que

2.1 Herramientas

las físicas se pueden modificar, además de ser el peón la clase que cuenta con la función `SetupPlayerInputComponent()` la cual nos permite acceder a la mencionada entrada del teclado.

2.1.2. Gitkraken:

Esta es una herramienta multiplataforma, es decir, se encuentra disponible para múltiples sistemas operativos la cual simplifica el uso de Git con el fin de mejorar la productividad. Este software nos permite crear repositorios, modificarlos y organizarlos de manera simple y visual. Este programa cuenta con una interfaz gráfica intuitiva y proporciona herramientas de resolución de conflictos de fusión de ramas, integración de plataformas de alojamiento y más funciones avanzadas de Git.

El principal uso que se le ha dado es la organización del proyecto en GitHub, dado que por limitaciones técnicas el trabajo se ha repartido en varias máquinas y era necesario migrar los avances entre las mismas.

2.1.3. Github:

Plataforma de control de versiones y colaboración que permite a programadores y desarrolladores de cualquier entorno almacenar su código y llevar un control del versionado del mismo. También facilita la participación de otros individuos en código público.

El uso de esta plataforma ha sido crucial para la organización y control de versiones de este proyecto por el hecho de tener que usar más de una máquina para completar ciertas tareas. Al mismo tiempo, ha facilitado la tarea de compartir código con el tutor.

2.1.4. ROS2:

ROS 2, también conocido como *Robot Operating System 2*, es un SDK (*System Development Kit*) open source, el cuál ofrece una plataforma estándar para desarrollar software de cualquier rama de la industria que implique el uso de robots. Este framework se desarrolló en 2007 por el Laboratorio de Inteligencia Artificial de Standford y su desarrollo ha continuado desde entonces facilitando el desarrollo de software para multitud de sistemas robóticos..

La versión de este framework con la que estamos trabajando es la versión *Humble*, la última versión publicada a la fecha de realización de este trabajo. Entrando a describir más específicamente en qué consiste este software, ROS se compone de 2 partes básicas, el sistema operativo ros, y ros-pkg, un conjunto de paquetes creados por la comunidad que implementan diversas funcionalidades como puede ser: localización, mapeo simultáneo, planificación, percepción y simulación...etc. Sin embargo, el uso principal de este conjunto de librerías es la transmisión de mensajes entre un controlador y la máquina en cuestión.

En este trabajo, nos planteamos hacer uso de ROS 2 para realizar el control del dron de forma “externa” y así simular un vuelo real. Siendo más explícitos comunicando un controlador que envíe instrucciones de forma externa al entorno de Unreal que muevan el dron en el escenario.

2.1.5. Aerostack2:

Este es un framework multiplataforma open source el cual se utiliza para el desarrollo de sistemas autónomos en entornos aeroespaciales, proporcionando una plataforma para diseñar, implementar y controlar drones así como otro tipo de sistemas autónomos aéreos. Este tiene como objetivo ayudar en el desarrollo de aplicaciones robóticas proporcionando una arquitectura modular y escalable, permitiendo la implementación de sistemas de navegación, percepción, control y comunicación.

Asimismo, cuenta con una API bien definida y componentes predefinidos que facilitan el desarrollo de diversos tipos de módulos. Algunas de sus características incluyen el uso de aprendizaje automático para la mejora de los sistemas autónomos así como ofrecer soporte a la planificación y coordinación de misiones complejas con varios vehículos.

En este trabajo se usará como complemento para el visualizador de vuelo desarrollado.

2.2. Estado del arte de los simuladores de drones

La simulación de drones es un campo relativamente nuevo que, sin embargo, ha experimentado un desarrollo exponencial en los últimos años. Durante este tiempo se han realizado significativos avances en el estudio de la física del movimiento de los drones, un aspecto esencial para su posterior simulación en entornos virtuales, que permiten explorar las posibles aplicaciones de estas versátiles máquinas. Como por ejemplo, cabe destacar su uso en seguridad[ano22b], mantenimiento de campos de placas solares[Mol19] e incluso entretenimiento con enjambres de cientos de estos pequeños robots[Rus21] .

Los drones son vehículos aéreos no tripulados que se crearon en un inicio con fines militares. Sin embargo, con el paso del tiempo se han encontrado una amplia variedad de usos. Por ello, simular el comportamiento de una máquina de este tipo se ha vuelto esencial, tanto para prever que movimientos será capaz de realizar según en qué condiciones se encuentre como para poder hacer pruebas sin el equipamiento real, evitando así posibles daños y costes. Con esto en mente, se crean los simuladores, que son entornos virtuales tridimensionales los cuales pueden recrear una gran variedad de situaciones.

En referencia a los simuladores, los últimos avances tecnológicos han potenciado su gran diversificación en multitud de aplicaciones como, aprender a controlar un dron, videojuegos de simulación de carreras de drones[ano22a], etc, básicamente orientadas a fines lúdicos. A nivel más profesional podemos encontrar al-

2.2 Estado del arte de los simuladores de drones

gunos como AirSim[Mic17], simulador de vuelo creado por Microsoft en el motor de juego de Unreal Engine (UE), o Flightmare[yl21], un simulador desarrollado para el motor de juego de Unity, más enfocados en aplicaciones industriales. Sin embargo, este tipo de simuladores basan sus físicas en el sistema Gazebo, que presenta ciertas limitaciones operativas en referencia al sistema de colisiones, lo que hace necesario nuevos desarrollos que mejoren estos aspectos.

La mayoría de los simuladores previamente descritos hacen uso de 2 SDK (Software Development Kit) externos, Gazebo y ROS/ROS2. El primero es un motor de físicas que gestiona todo lo relacionado con la física que interacciona con el dron, ya sea su movimiento, velocidad...etc. El segundo, ROS, es un software que se centra en el intercambio de mensajes, más concretamente, se encarga de enviarle las instrucciones al controlador del dron para manejar al mismo. Le envía datos como por ejemplo el modo de vuelo, velocidad y demás. Estos modos de vuelo y datos pueden variar dependiendo de que tipo de dron y sobre que tipo de software este construido.

Así pues, este trabajo tiene como objetivo crear un prototipo de un simulador de vuelo de drones en el entorno gráfico de Unreal Engine 5 realizando una integración con la librería de comunicación de C, ROS2, para así crear un sistema de manejo automático de la trayectoria del dron. La implementación de la gran mayoría del proyecto se realizará en C++. Al mismo tiempo, al crear este simulador en UE5, se pretende resolver algunas de las limitaciones asociadas al uso de Gazebo como motor de físicas, aprovechando la infraestructura proporcionada por UE5.

3. Desarrollo

3.1. Descripción General

Como se mencionó en la introducción, el desarrollo de esta aplicación será principalmente en C++ dado que es el lenguaje en el que se programa en Unreal Engine 5 y ROS2.

La metodología seguida a lo largo del trabajo ha sido una metodología ágil informal dado que este trabajo ha sido realizado por 2 personas, el tutor tomando el rol del cliente y el alumno tomando el rol de desarrollador. Es debido a esta forma de trabajo que el desarrollo se basa en sprints o iteraciones. Estas iteraciones integran los progresos realizados siguiendo los hitos marcados por el tutor. El comienzo de una iteración se encuentra marcado por la definición de un requisito por parte del tutor y su fin lo marca el resultado obtenido por el desarrollo del alumno que ha de satisfacer los objetivos definidos por el tutor.

A continuación se detallan los diferentes hitos fijados para este TFG.

3.2. Hito 1: Exploración del movimiento de un objeto en Unreal

3.2.1. Descripción:

Par implementar un dron, el primer paso sería conocer la manera de proporcionar movimiento a objetos dentro del motor de juego. El objetivo de este hito es desarrollar una manera estable con la que poder mover objetos dentro de Unreal.

3.2.2. Desarrollo:

Con el fin de lograr este hito, se desarrolló un componente de escena que pudiera ser añadido a actores. La idea se basa en poder tener un módulo que implemente el movimiento y se pueda migrar a otros actores para que se comporten de la misma manera.

Haciendo uso de estos componentes diseñamos una primera iteración para estudiar el movimiento en los actores y planificar la futura implementación del movimiento de un dron. Tras investigar y revisar tutoriales de diversas fuentes[Eng23] decidí crear un componente de escena que se encargara de modificar la localización de una `mesh` (malla). Es decir, crear un componente de escena que fuera el raíz de la jerarquía que moviera un modelo 3D. En este caso, un cubo cuyo movimiento se basa en un vector de posición al que se le pueden ajustar

las componentes tanto desde el editor, desde un blueprint como desde el propio código fuente.

A continuación abordamos la cuestión de las colisiones, añadiendo la propiedad comúnmente conocida como hitbox (caja de colisiones). Para esto simplemente se añade un componente de escena de tipo hitbox, se ajusta al tamaño de la mesh elegida y se configura un gráfico de eventos para que cuando el jugador entre en contacto con el objeto este empiece a moverse. Asimismo, se incluye otra hitbox para delimitar un área amplia alrededor de la malla para que en el caso de que detectara que el jugador abandona el área el objeto volviera a su posición original.

Atendiendo al código desarrollado para este hito se creó la clase move, clase que genera un componente de escena compuesto por dos ficheros un header y el .cpp conteniendo el código fuente. En estos, a parte de las funciones características de Unreal se crearon otras 3 funciones diferentes para configurar el movimiento del actor en el que se situase este componente. Las funciones son EnableMovement(), ResetMovement() y SetMoveDirection().

Listing 3.1: Definición de funciones en el header de move

```

1 public:
2     // Sets default values for this component's properties
3     Umove();
4
5     UFUNCTION(BlueprintCallable)
6     void EnableMovement(bool shouldMove);
7
8     UFUNCTION(BlueprintCallable)
9     void ResetMovement();
10
11    UFUNCTION(BlueprintCallable)
12    void SetMoveDirection(int Direction);

```

Estas 3 funciones se encargan de configurar el movimiento, siendo este realizado en la función TickComponent(). Entrando en más detalle, la función EnableMovement() se encarga de activar el movimiento en base a un boolean que se vuelve verdadero cuando el jugador colisiona con la hitbox del actor. La función ResetMovement() reinicia el movimiento del actor, esta función es declarada como *blueprint callable*, por lo que solo haremos uso de la misma desde el editor del blueprint. Finalmente la función SetMoveDirection() como indica su nombre, cambiará la dirección del movimiento ajustando un entero que se usará en la función que realice el movimiento.

Listing 3.2: Funciones EnableMovement(), ResetMovement() y SetMoveDirection()

```

1 void Umove::EnableMovement(bool shouldMove)
2 {
3     // Assign value and set correct tick enable state
4     MoveEnable = shouldMove;
5     SetComponentTickEnabled(MoveEnable);
6 }
7
8 void Umove::ResetMovement()

```

3.2 Hito 1: Exploración del movimiento de un objeto en Unreal

```
9  {
10     //Clear distance and set to origin
11     CurDistance = 0.0f;
12     SetRelativeLocation(StartRelativeLocation);
13 }
14
15 void Umove::SetMoveDirection(int Direction)
16 {
17     MoveDirection = Direction >= 1 ? 1 : -1;
18 }
```

Al mismo tiempo es importante comentar el código implementado en las funciones de `BeginPlay()` y `TickComponent()`. En la primera configuramos valores de inicio para algunas de las variables que necesitaremos para componer el movimiento del actor. Se configura la posición inicial del actor almacenando el vector en la variable `StartRelativeLocation`, a continuación, se inicializa el vector llamado `MoveOffsetNorm` al valor de las coordenadas a las que se desea mover el actor e inmediatamente se normaliza para obtener un movimiento fluido más adelante. Al mismo tiempo se configura la distancia máxima a la que se puede mover el actor y finalmente se llama a la función de Unreal `SetComponentTickEnabled()` para confirmar que se pueda usar la función de `TickComponent()` en este actor.

Listing 3.3: Función `BeginPlay()`

```
1 void Umove::BeginPlay()
2 {
3     Super::BeginPlay();
4
5     // Set start location
6     StartRelativeLocation = this->GetRelativeLocation();
7
8     //Compute normalized movement
9     MoveOffsetNorm = MoveOffset;
10    MoveOffsetNorm.Normalize();
11    MaxDistance = MoveOffset.Size();
12
13    //Check if ticking is required
14    SetComponentTickEnabled(MoveEnable);
15 }
16 }
```

Para terminar, la función `TickComponent()`, es donde se realiza el movimiento. Nada más entrar a la función se comprueba si la colisión que inicia el movimiento ha ocurrido mediante la variable `MoveEnable`, en caso afirmativo se realiza un cálculo para obtener la distancia actual recorrida y se comprueba si ha alcanzado el máximo o un mínimo (siendo este el que desee el programador, aquí está configurado a 0 para que no atraviese el suelo). En caso de que no halla alcanzado ninguno de los dos, se pasa a ajustar la posición teniendo en cuenta su posición inicial, el vector normalizado y la distancia recorrida. En caso de haber alcanzado alguno de los límites, se siguen los siguientes pasos: se cambia la dirección de movimiento, se lanza el evento `OnEndpointReached` el cual definimos como `BlueprintAssignable` en el fichero de cabecera, ajustamos la distancia actual mediante la función de librería `FMath`, `Clamp()`, finalmente se ajusta la posición nueva del actor.

Listing 3.4: Función TickComponent()

```

1 void Umove::TickComponent(float DeltaTime, ELevelTick TickType,
2                           FActorComponentTickFunction* ThisTickFunction)
3 {
4     Super::TickComponent(DeltaTime, TickType, ThisTickFunction);
5
6     //Set the current distance
7     if(MoveEnable)
8     {
9         CurDistance += DeltaTime * Speed * MoveDirection;
10        if(CurDistance >= MaxDistance || CurDistance <= 0.0f){
11            //Inver direction
12            MoveDirection *= -1;
13
14            //Fire event
15            OnEndpointReached.Broadcast(CurDistance >= MaxDistance);
16
17            //Clamp distance
18            CurDistance = FMath::Clamp(CurDistance, 0.0f, MaxDistance);
19        }
20
21
22        // Compute and set current location
23        SetRelativeLocation(StartRelativeLocation + MoveOffsetNorm * CurDistance);
24    }
}

```

Sin embargo, no todo el comportamiento de este actor se rige por el código. Para este hito también se hace uso de la programación en blueprint basada en gráficos de eventos. Es aquí donde una vez creado el actor en Unreal y añadirle el componente de escena creado previamente, se ha de configurar el siguiente gráfico de eventos:

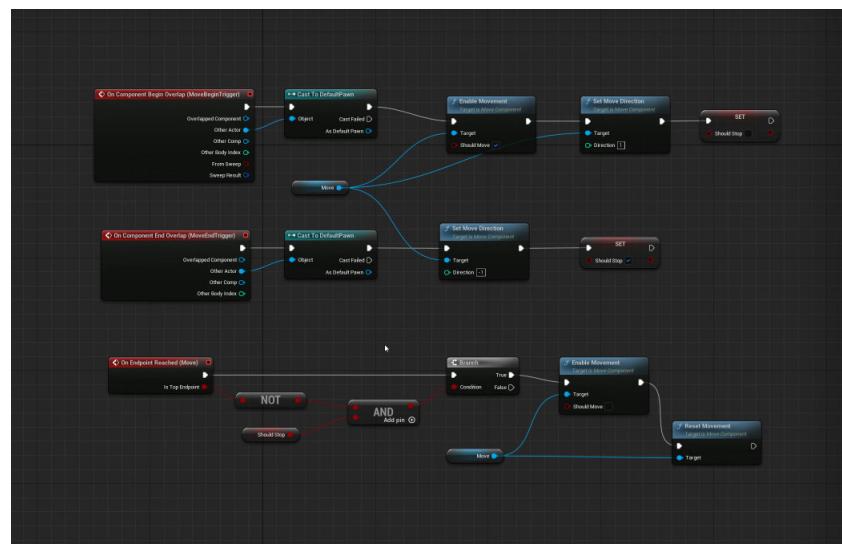


Figura 3.1.: Gráfico de eventos de activación de movimiento por colisión de hitbox

En este podemos ver que es donde se configuran las colisiones y qué funciones son llamadas en base a los eventos que ocurren en la escena de Unreal. En el evento de Begin Overlap es donde se detecta la colisión inicial y comienza el

3.2 Hito 1: Exploración del movimiento de un objeto en Unreal

movimiento llamando a `EnableMovement()`. En la misma línea podemos ver que se llama también a `SetMoveDirection()` donde en base al evento que se halla detectado el valor dado a la dirección varia, si estamos en la línea de `Begin Overlap` el valor es 1 y si estamos en la línea de `End Overlap` es -1 para realizar el movimiento a la inversa volviendo al punto de origen. Finalmente podemos ver como al final de la línea de `On Endpoint Reached` se llama a `ResetMovement()` para que en caso de que se llegue al final del movimiento, se reinicie la posición del actor.

El código y el gráfico de eventos fue obtenido de la guía creada por Lötwig Fusel [Fus22a][Fus22b]

Posteriormente, tras este primer acercamiento a los actores y componentes, decidí realizar pruebas para intentar implementar un modo de vuelo típico de los drones, el modo *Hover*. Este es un modo simple en el que el dron se eleva hasta una altura pre establecida y la mantiene hasta que se le envíe una señal para moverse o volver a su posición original.

Esta prueba es muy similar a la anterior dado que también se basa en un componente de escena, sin embargo, hay algún cambio respecto al funcionamiento. El código para modificar la posición sigue siendo el mismo, solo que en este caso, podemos modificar únicamente la altura máxima que queremos que alcance el dron y la velocidad del mismo. Esto es observable en la definición de las variables, donde solo permitimos una entrada variable que es la coordenada en el eje Z que luego se incluye en un vector para realizar el movimiento como en el componente `move`.

Listing 3.5: Definición de variables del componente HoverComponent

```
1 private:
2
3     //Toggle para saber si iniciar el hover o no
4     UPROPERTY(EditAnywhere)
5     bool TriggerHover = true;
6
7     //Toggle para saber si resetear el hover
8     UPROPERTY(EditAnywhere)
9     bool Reset = true;
10
11    //Altura en la que queremos el dron
12    UPROPERTY(EditAnywhere)
13    float Height = 500.0f;
14
15    //Velocidad a la que queremos que se mueva
16    UPROPERTY(EditAnywhere)
17    float Speed = 120.0f;
18
19    //Ubicaciones computadas o a computar
20    FVector StartRelativeLocation;
21    FVector HeightNorm;                                //Variable para
22    // normalizar el vector d ela posicion final y poder mover el dron de "
23    // poco en poco"
24    float MaxHeight = 0.0f;
25    float ActualHeight = 0.0f;
26};
```

También se implementaron dos funciones, Hover() para comenzar el vuelo y ResetPosition() para volver a la posición original. Cabe mencionar que esta vez no se ha implementado ningún gráfico de eventos dado que la interacción de esta prueba se basa en el envío de la señal de activación o de reset. Estas señales se mandan desde el editor, pudiendo encontrar dos botones en el mismo una vez se compila este componente. Si se selecciona la opción de hover en el editor, comenzará el movimiento. Si se selecciona la de reset se volverá al punto de origen.

Listing 3.6: Funciones Hover() y ResetPosition()

```

1 void UHoverComponent::Hover(float DeltaTime)
2 {
3     //Establecemos la altura actual
4     ActualHeight += DeltaTime * Speed * 1;
5     //Computamos y actualizamos la ubicacion en caso de no haber llegado a la
6     //altura maxima
7     if(ActualHeight <= MaxHeight || ActualHeight <= 0.0f){
8         SetRelativeLocation(StartRelativeLocation + HeightNorm *
9             ActualHeight);
10    }
11 }
12
13 void UHoverComponent::ResetPosition()
14 {
15     FVector start;
16     start.X = 0;
17     start.Y = 0;
18     start.Z = 40.0f;
19     SetRelativeLocation(start);
20 }
```

3.2.3. Resultados:

Como resultado de este hito podemos encontrar los ficheros adjuntados en el anexo en la página 37. Estos ficheros configuran los componentes de escena move y HoverComponent los cuales cumplen el propósito de este hito, movimiento de objetos dentro de Unreal.

Esta forma de mover los elementos nos será útil para futuras iteraciones pues es una forma simple y eficaz de cambiar la posición de objetos dentro de este editor. Sin embargo, hacerlo mediante componentes pensamos que no es suficientemente eficaz dado que al ser un elemento que se ha de añadir a un actor, ofrece limitaciones a la hora de configurar inputs por parte del usuario además de ofrecer limitadas interacciones con el propio actor y otros posibles componentes. Así pues, en futuras iteraciones haremos uso de peones, más concretamente el objetivo será construir un peón propio desde cero.

3.3 Hito 2: Importación de mallas

3.3. Hito 2: Importación de mallas

3.3.1. Descripción:

Además de movimiento, para poder realizar un simulador también es importante el apartado estético. Por ello el objetivo de este hito es investigar el funcionamiento de la importación de mallas para conseguir un modelo tridimensional de un cuadricóptero dentro del propio entorno de Unreal.

3.3.2. Desarrollo:

Este tipo de modelos tridimensionales, también conocidos como mallas o *mesh* son muy importantes dentro de Unreal ya que es gracias a estos que el editor es capaz de simular físicas y colisiones basándose en hitboxes ajustadas a estas mallas.

Para empezar busqué en internet un modelo 3D gratuito de un cuadricóptero [referencia a la página del modelo 3D del dron]. Una vez descargado faltaba importarlo a la carpeta de contenidos de Unreal Engine para su uso. Cabe mencionar que estos modelos 3D han de ser exportados de sus respectivas aplicaciones en formato .fbx ya que este es el que reconoce Unreal. El proceso de importación es muy simple, consiste en arrastrar el archivo al editor y tras una simple ventana de configuración ya estaría disponible el modelo para su uso.

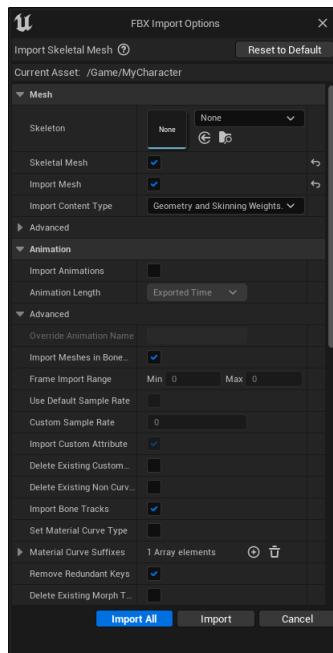


Figura 3.2.: Ventana de configuración de importaciones UE5

Una vez importada la malla, para utilizarla basta con arrastrarla del cajón de contenidos a la escena.

Si se desea, Epic Games también cuenta con una plataforma en la que los usuarios pueden subir a una tienda on-line sus modelos 3D ya sean escenarios u objetos. Una posible limitación es que la mayoría del contenido es de pago.

Es importante mencionar que una vez se importa una malla es posible modificarla dentro del editor de blueprints para reajustar sus valores, materiales, texturas...etc.

3.3.3. Resultados:

Como resultado de la malla elegida obtenemos el siguiente modelo 3D en nuestra escena de Unreal

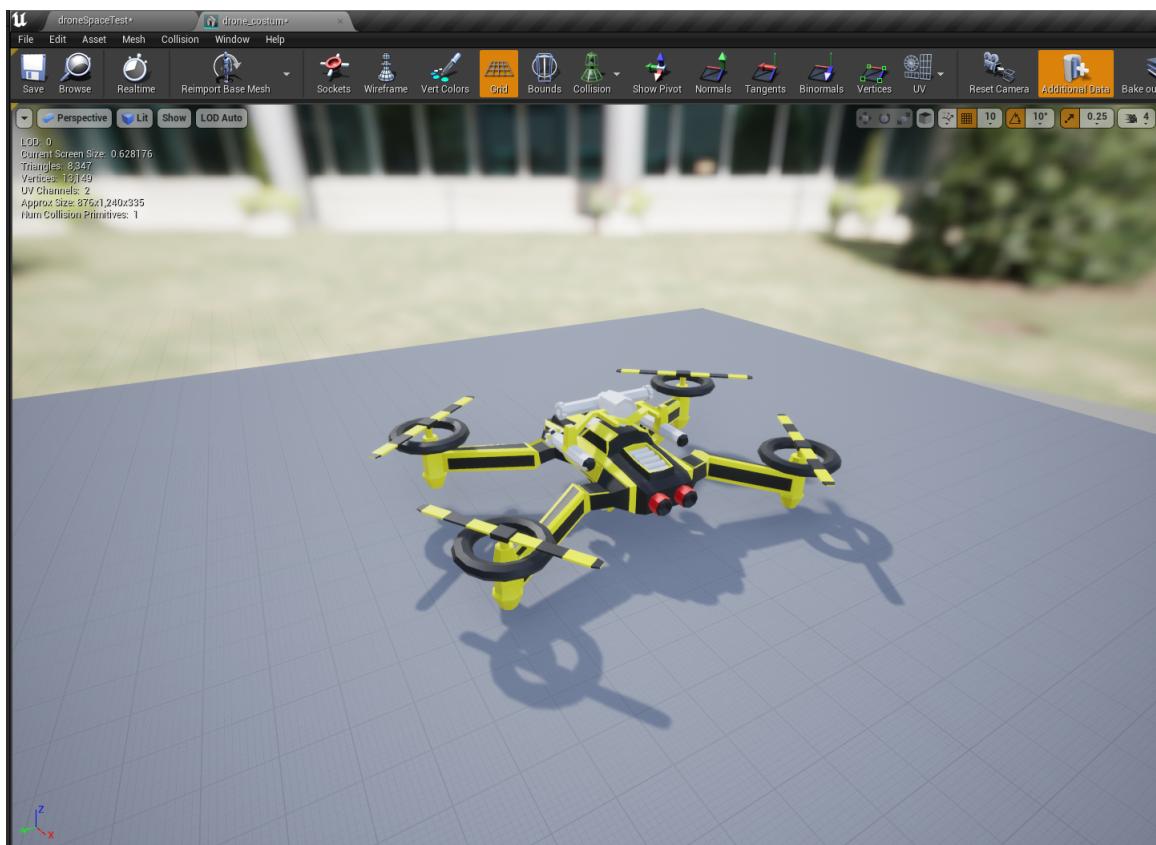


Figura 3.3.: Mesh del dron en el entorno de Unreal

Con esto quedaba solventado el problema de poder disponer de un modelo tridimensional de un dron. Un objetivo de futuro será determinar si Unreal genera las colisiones automáticamente al proporcionarle una hitbox al modelo o si estas han de ser programadas desde 0.

3.4. Hito 3: Implementación de un control diferencial de la velocidad

3.4.1. Descripción:

El objetivo de este hito es codificar las funciones necesarias para implementar un control diferencial de la velocidad en un peón de Unreal y controlar la velocidad con teclas para estudiar el funcionamiento de la entrada por teclado. Este tipo de controlador es muy útil ya que es una muy buena aproximación de como se implementa realmente el movimiento de los drones.

3.4.2. Desarrollo:

Como se menciona en la descripción de este hito, se va a crear un peón. Para la creación de un control diferencial de la velocidad es necesario contar con 3 vectores, uno para establecer la velocidad deseada, otro para definir la velocidad real del dron y un tercero que se encarga de calcular un delta con la velocidad real actual del dron, la velocidad objetivo y el tiempo transcurrido.

Listing 3.7: Vectores utilizados

```
1  UPROPERTY(EditAnywhere)
2  FVector realVelocity;
3
4  UPROPERTY(EditAnywhere)
5  FVector deltaVelocity;
6
7  UPROPERTY(EditAnywhere)
8  FVector targetVelocity;
```

Para realizar los cálculos necesarios utilizaremos la función Tick() dada por Unreal dentro del esqueleto de la clase peón. En esta función se hará uso de la variable DeltaTime para calcular la velocidad real y en un futuro también se utilizará para mover el dron y calcular su posición. Asimismo, dada la naturaleza de Tick() esta será la función con más carga computacional ya que se encargará de realizar todos los cálculos o realizar llamadas a las funciones que los hagan, en este caso a CalculateDelta() y CalculateVelocity().

Listing 3.8: Función Tick()

```
1 void ADronePawn::Tick(float DeltaTime)
2 {
3     Super::Tick(DeltaTime);
4
5     realVelocity = GetActorLocation() * DeltaTime;
6     CalculateDelta();
7     CalculateVelocity(); //CalculateVelocity
8 }
```

Las funciones CalculateDelta() y CalculateVelocity() realizan las funciones que sus nombres describen, CalculateDelta() calcula el delta de la velocidad siguiendo la ecuación de un movimiento rectilíneo uniformemente acelerado.

Listing 3.9: Función CalculateDelta()

```

1 void ADronePawn::CalculateDelta()
2 {
3     deltaVelocity = (targetVelocity - realVelocity) * velocityConstant;
4 }
```

Y la función CalculateVelocity() realiza una serie de comprobaciones. Estas consisten en comprobar si el delta de la velocidad calculado supera unos límites que se establecen en el fichero de cabecera como medida de seguridad. Si el delta calculado supera los límites entonces se ajusta al máximo o mínimo, en caso contrario ajustamos la velocidad real sumándole ese delta hallado previamente.

Listing 3.10: Función CalculateVelocity()

```

1 void ADronePawn::CalculateVelocity()
2 {
3     if(deltaVelocity.X >= upperLimit)
4     {
5         deltaVelocity.X = upperLimit;
6     }
7     if(deltaVelocity.X <= lowerLimit)
8     {
9         deltaVelocity.X = lowerLimit;
10    }
11    if(deltaVelocity.Y >= upperLimit)
12    {
13        deltaVelocity.Y = upperLimit;
14    }
15    if(deltaVelocity.Y <= lowerLimit)
16    {
17        deltaVelocity.Y = lowerLimit;
18    }
19    if(deltaVelocity.Z >= upperLimit)
20    {
21        deltaVelocity.Z = upperLimit;
22    }
23    if(deltaVelocity.Z <= lowerLimit)
24    {
25        deltaVelocity.Z = lowerLimit;
26    }
27
28    realVelocity += deltaVelocity;
29 }
```

Listing 3.11: Límites establecidos para el delta

```

1 UPROPERTY(EditAnywhere)
2 float upperLimit = 1000.0f;
3
4 UPROPERTY(EditAnywhere)
5 float lowerLimit = -1000.0f;
```

En BeginPlay() para este hito no realizamos ninguna función relevante a parte de la inicialización de un vector.

Especial mención a la función SetupPlayerInputComponent() ya que en esta función es donde configuraremos los inputs del teclado que tenemos como objetivo. Para configurar los inputs del teclado se han de seguir tres pasos: el primero ha-

3.4 Hito 3: Implementación de un control diferencial de la velocidad

ce relación a la configuración interna del proyecto, donde hemos de configurar los *keybinds* en la pestaña de inputs dentro de ajustes de proyecto.

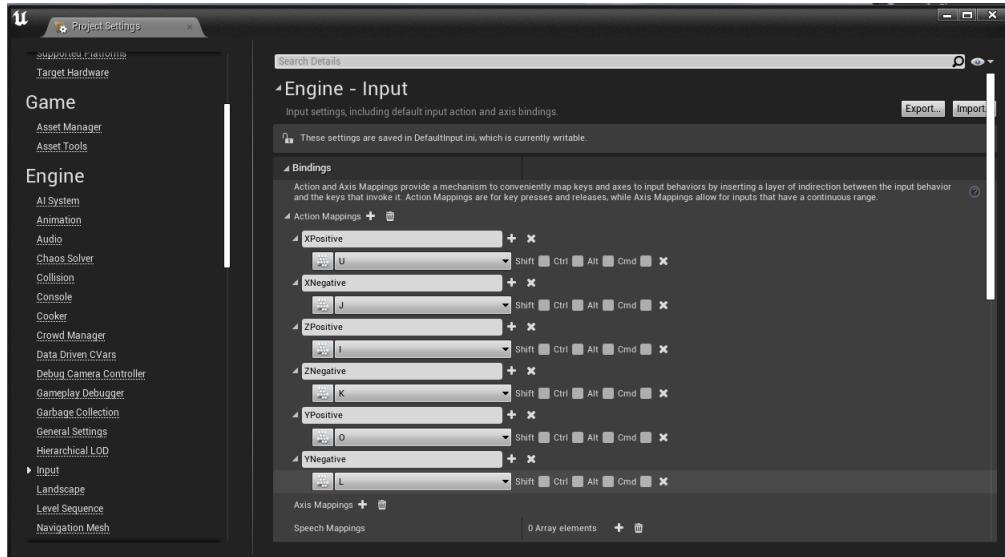


Figura 3.4.: Asociación de teclas en la configuración del proyecto

Estos posteriormente se relacionan con el segundo paso, donde habremos de ir al código y dentro de la función `SetupPlayerInputComponent()` hacer uso de la función `BindAction()` configurando sus parámetros para que coincidan con los puestos en el editor.

Listing 3.12: Configuración de la función `SetupPlayerInputComponent()`

```
1 //Setting up bindings to add or subtract from the targetVelocity
2 PlayerInputComponent->BindAction("XPositive", IE_Pressed, this, &ADronePawn
3     ::ModifySpeedX);
4 PlayerInputComponent->BindAction("XNegative", IE_Pressed, this, &ADronePawn
5     ::ModifySpeedNegativeX);
6 PlayerInputComponent->BindAction("YPositive", IE_Pressed, this, &ADronePawn
7     ::ModifySpeedY);
8 PlayerInputComponent->BindAction("YNegative", IE_Pressed, this, &ADronePawn
9     ::ModifySpeedNegativeY);
10 PlayerInputComponent->BindAction("ZPositive", IE_Pressed, this, &ADronePawn
11     ::ModifySpeedZ);
12 PlayerInputComponent->BindAction("ZNegative", IE_Pressed, this, &ADronePawn
13     ::ModifySpeedNegativeZ);
```

Al mismo tiempo a las teclas seleccionadas hay que asignarles funcionalidades que han de definirse en la llamada a `BindAction()`, en nuestro caso son funciones que modificarán la velocidad objetivo en cada eje.

Listing 3.13: Funciones asociadas a teclas

```

1     void ModifySpeedX();
2     void ModifySpeedNegativeX();
3     void ModifySpeedY();
4     void ModifySpeedNegativeY();
5     void ModifySpeedZ();
6     void ModifySpeedNegativeZ();
7     void CalculateDelta();
8     void CalculateVelocity();

```

Finalmente para terminar de configurar la entrada por teclado, el tercer paso requiere modificar los ficheros del juego base, los ficheros `nombreDeProyectoGameModeBase.cpp` y `nombreDeProyectoGameModeBase.h` para que tomen como peón predeterminado el que hemos creado. Para esto basta con crear un constructor en el fichero de cabecera y en el `.cpp` dentro de la función hacer uso de la variable `DefaultPawnClass` escribiendo la siguiente línea "`DefaultPawnClass = nombre_de_su_peón`".

3.4.3. Resultados:

Como resultado de este hito podemos encontrar los ficheros adjuntados en el anexo en la página 41.

Asimismo al ejecutar el nivel y seleccionar el peón, observaremos en la pestaña de detalles que si pulsamos las teclas, los valores de los vectores varían acorde las ecuaciones y funciones implementadas siguiendo el esquema de un control diferencial de la velocidad.

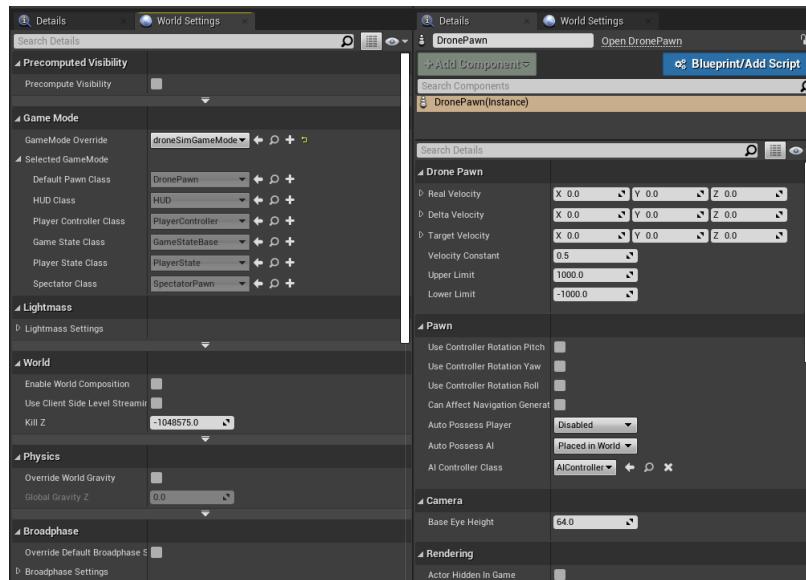


Figura 3.5.: Valores del editor del modo de juego y los vectores asociados al peón

Ya configurado y testado el correcto funcionamiento del modelo, el siguiente paso es modificar la posición del propio peón con las velocidades calculadas. Tras esta última prueba concluimos que el movimiento del dron basado en este control diferencial es lo necesario para simular correctamente el movimiento y poder

3.4 Hito 3: Implementación de un control diferencial de la velocidad

realizar una integración con ROS2, ya que pretendemos que el movimiento se gobierne mediante la velocidad que se enviará mediante los servicios de mensajería ofrecidos por este framework.

3.5. Hito 4: Desarrollo de un sistema pub/sub de ROS2

3.5.1. Descripción:

En la realidad, el control remoto de un dron es posible gracias a la biblioteca de C/C++ previamente descrita, ROS2. Es por esto por lo que el objetivo de este hito es crear un sistema publicador-suscriptor el cual permita crear una librería del suscriptor para, en un futuro, poder implementar su funcionalidad en un peón de Unreal.

Como especificaciones del sistema, el publicador ha de poder leer las coordenadas de la entrada estandar, convertirlas en un vector y publicarlas para ser recibidas por el suscriptor. Asimismo, ha de poder crearse una librería a partir del suscriptor, teniendo en mente la futura implementación de la librería en Unreal.

3.5.2. Desarrollo:

La tarea de creación del suscriptor fue ampliamente simplificada gracias al trabajo de Juan Pares Guillen[Tap23b] y Santiago Tapia, quienes adaptaron un suscriptor para su trabajo el cual he podido importar a mi proyecto y hacer uso del mismo puesto que la funcionalidad que buscaba implementar es similar. Este suscriptor cuenta con 3 funciones, `start()`, `update()` y `end()`. Cada una de estas funciones tiene un propósito y son las que más adelante importaremos en el peón de UE5:

- `start()`: Esta función se encarga de inicializar el suscriptor y suscribirse al canal (*topic*) creado por el publicador. Asimismo, configurará su función de *callback* a `update()`, encargada de recibir los mensajes.
- `update()`: Esta es la función de *callback*, la cual recibirá los mensajes que se vayan publicando. Esta función será ejecutada según el intervalo de tiempo puesto en su inicialización dentro de la función `start()`.
- `end()`: Se encarga de destruir el suscriptor y liberar la memoria ocupada.

Listing 3.14: Funciones `start()`, `update()` y `end()`

```

1 void start()
2 {
3     int argc = 1;
4     char name[] = "subscriber";
5     char *argv[] = {name};
6
7     rclcpp::init(argc, argv);
8     executor = std::make_shared<Executor>();
9     node = std::make_shared<MinimalSubscriber>();
10    executor->add_node(node);
11 }
12 // PILA
13 // OPCION 1: devolver void* en start, recibirlo en el update y hacer cast a
14 //             MinimalSubscriber
15 // OPCION 2: cola como atributo general, al principio del fichero
16 int update(vector3_transfer *ptr)

```

```

17 {
18     executor->spin_some();
19     if (!node->dequeue.empty())
20     {
21         auto msg = node->dequeue.front();
22         node->dequeue.pop_front();
23         ptr->x = msg->x;
24         ptr->y = msg->y;
25         ptr->z = msg->z;
26         return 1;
27     }
28     return 0;
29 }
30
31 void end()
32 {
33     node.reset();
34     rclcpp::shutdown();
35 }
```

Esta clase hace uso del tipo de mensaje Vector3 de la librería geometry_msgs, tipo propio de ROS2 que ha de incluirse en la clase y posteriormente ha de indicarse en el CMAKE para su correcta compilación.

Listing 3.15: Especificación de compilación del suscriptor

```

1 add_executable(listener src/listener_member_function.cpp)
2 ament_target_dependencies(listener rclcpp geometry_msgs)
```

A continuación, en lo relacionado al publicador de ROS2 se toma como base el publicador de ejemplo que ROS2 propone en sus tutoriales[org23d]. La base es la misma, solo que se hace uso de un tipo de mensaje diferente, el Vector3, igual que en el suscriptor.

Listing 3.16: Especificación de compilación del publicador

```

1 add_executable(talker src/CoordinatesPublisher.cpp)
2 ament_target_dependencies(talker rclcpp geometry_msgs)
```

Una vez hecho esto, el otro cambio importante a realizar es la lectura de los datos, para lo cual se hace uso de la lectura por entrada estandar. Este método es simple, solo que hemos de tener en cuenta que hay que transferirle los datos en la línea de ejecución. En este caso se hace mediante el comando cat y el uso de una tubería para pasar las coordenadas redactadas en un fichero “coordinates.txt” que se encuentra en la carpeta de ejecución del publicador.

Ahora necesitamos poder compilarlos, para lo cual seguimos las indicaciones de la guía oficial de ROS2[org23b]. Puesto que nuestros programas son diferentes a los creados en el tutorial hemos de realizar modificaciones en el CMAKE para conseguir compilar con éxito. Dado que usamos un tipo de mensajes diferente al indicado en la guía, tenemos que especificar que se busque el paquete donde se encuentra el tipo de mensaje Vector3, el paquete geometry_msgs

Listing 3.17: Paquetes necesarios

```
1 find_package(ament_cmake REQUIRED)
2 find_package(rclcpp REQUIRED)
3 find_package(std_msgs REQUIRED)
4 find_package(geometry_msgs REQUIRED)
```

Finalmente es necesario añadir también un `add_library` para crear así la librería del suscriptor que queremos utilizar en Unreal.

Listing 3.18: Creación de la librería del suscriptor

```
1 add_library(listenerlib SHARED src/listener_member_function.cpp)
2 ament_target_dependencies(listenerlib rclcpp geometry_msgs)
```

Tras estos cambios al CMAKE, podemos compilar el proyecto con la herramienta `colcon` de ROS2 creando así la librería del suscriptor y consecuentemente un ejecutable del publicador.

3.5.3. Resultados:

Los resultados de este hito se incluyen en los archivos adjuntos en el anexo en la página 44.

Este desarrollo resuelve lo planteado en los objetivos de este hito cuyo fin era obtener un sistema publicador suscriptor, teniendo en cuenta la futura implementación del mismo en Unreal. Tras esto, el siguiente paso a seguir es la gestión de la librería obtenida dentro de un peón en Unreal.

3.6. Hito 5: Desarrollo del peón en UE5

3.6.1. Descripción:

Este hito se centra en la creación de un peón desde 0 que implemente el sistema publicador-suscriptor descrito anteriormente y pueda moverse en el entorno de Unreal.

Este peón será la pieza clave de un visualizador de vuelo que sentará la base para futuras iteraciones de este proyecto con el objetivo de construir un simulador.

3.6.2. Desarrollo:

Para conformar este peón haremos uso de una variedad de funciones, algunas previamente mencionadas pues son las generadas automáticamente por Unreal, otras auxiliares creadas por nosotros y otras importadas desde la librería del suscriptor. Nos encontramos una vez más con las funciones `BeginPlay()`, `Tick()`, el constructor del propio peón, y además haremos uso de `EndPlay()` solo que esta la sobrescribiremos. En cada una de estas podremos encontrar otras 3 funciones auxiliares las cuales implementan las funcionalidades deseadas. Al

3.6 Hito 5: Desarrollo del peón en UE5

mismo tiempo se hará uso de las funciones importadas de la librería en 3 de las 4 funciones mencionadas anteriormente.

Una breve descripción de las funciones es la siguiente:

- BeginPlay(): en esta veremos la función `dlImport()`, función auxiliar la cual se encarga de gestionar la apertura de la librería dinámica que contiene las funciones necesarias para hacer uso del suscriptor. Dentro de esta función veremos un `dlopen()` con su respectivo control de fallos y la importación de las funciones de la librería a este fichero mediante el uso de las funciones `dlsym()`. La importación de estas funciones se consigue gracias a que almacenamos las referencias a estas en punteros a función definidos mediante un `typedef` previamente en el fichero de cabecera. Cabe mencionar que en esta función también se realiza la llamada `start()` función importada de la librería.

Listing 3.19: Definición de los punteros a función en el fichero de cabecera

```
1      //Start function, sets the listener up
2      typedef void (*fun_start)();
3      fun_start start;
4
5      //struct which holds the information needed in the update function
6      struct vector3_transfer
7      {
8          float x, y, z;
9      };
10     typedef struct vector3_transfer vector3_transfer;
11
12     //Update function will be called in the tick, used for receiving info
13     //from talker/publisher
14     typedef int (*fun_update)(vector3_transfer *);
15     fun_update update;
16
17     //End function for closing the library
18     typedef void (*fun_end)();
19     fun_end end;
```

Listing 3.20: Función BeginPlay()

```
1 void ADronePawnCom::BeginPlay()
2 {
3
4     Super::BeginPlay();
5     //We call the dlImport to open the ros2 library
6     dlImport();
7     //Here since unreal only executes this begin play once and at the start of
7     //the level
8     //we call the start function which sets up the listener in order to
8     //receive the
9     //coordinates from the ROS2 publisher/talker
10    start();
```

Listing 3.21: Función dlImport()

```
1 void ADronePawnCom::dlImport()
2 {
3     //Here i'll set the call to the dynamic library created for the ros2
3     //communication
```

```

4         //And all the functions neccesary in order to set up the functioning
5         listener
6         handle = dlopen("/home/daniel/ros2_ws/build/droneCom/liblistenerlib.so",
7                         RTLD_LAZY);
8
9         if (!handle)
10        {
11            /* fail to load the library */
12            fprintf(stderr, "Error: %s\n", dlerror());
13            return;
14        }
15
16        //Once the library is open, we take the finctions we want to use, in this
17        //case, start, update and end
18
19        void* temp1 = dlsym(handle, "start");
20        start = (fun_start)temp1;
21
22        if (!start)
23        {
24            /* no such symbol */
25            fprintf(stderr, "Error: %s\n", dlerror());
26            dlclose(handle);
27            UE_LOG(LogTemp, Error, TEXT("Error a la hora de cargar start"));
28            return;
29        }
30
31        void* temp2 = dlsym(handle, "update");
32        update = (fun_update)temp2;
33
34        if (!update)
35        {
36            /* no such symbol */
37            fprintf(stderr, "Error: %s\n", dlerror());
38            dlclose(handle);
39            UE_LOG(LogTemp, Error, TEXT("Error a la hora de cargar update"));
40            return;
41        }
42
43        void* temp3 = dlsym(handle, "end");
44        end = (fun_end)temp3;
45
46        if (!end)
47        {
48            /* no such symbol */
49            fprintf(stderr, "Error: %s\n", dlerror());
50            dlclose(handle);
51            UE_LOG(LogTemp, Error, TEXT("Error a la hora de cargar end"));
52            return;
53        }
54        else{
55            UE_LOG(LogTemp, Warning, TEXT("End importado"));
56        }
57    }

```

- El constructor: aquí podremos encontrar el método auxiliar `setComponents()` que como indica su nombre genera los componentes que irán colgados de este peón y podremos ver más adelante en la jerarquía dentro de su blueprint. Los componentes aquí generados son: una cámara para poder ver la malla del dron, un `SpringArmComponent`, el cual nos ayudará a configurar la posición relativa de la cámara respecto del dron, un `RootComponent`, del que

3.6 Hito 5: Desarrollo del peón en UE5

colgar el resto de los componentes, y un MeshComponent para poder darle un cuerpo al dron.

Listing 3.22: Constructor de la clase

```
1 ADronePawnCom::ADronePawnCom()
2 {
3     // Set this pawn to call Tick() every frame. You can turn this off to
4     // improve performance if you don't need it.
5     PrimaryActorTick.bCanEverTick = true;
6     //We call the setComponents function to set up all the pawns components
7     setComponents();
}
```

Listing 3.23: Función SetComponents()

```
1 void ADronePawnCom::setComponents()
2 {
3     //setup rootcomponent
4     RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("RootComponent"));
5     //Setup StaticMeshComponent
6     MeshComponent = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("DronMesh"));
7     MeshComponent->SetupAttachment(RootComponent);
8
9
10    //section for configuring the spring arm and camera attached to it
11    arm = CreateDefaultSubobject<USpringArmComponent>(TEXT("SpringArm"));
12    arm->SetupAttachment(RootComponent);
13
14    camera = CreateDefaultSubobject<UCameraComponent>(TEXT("CameraComponent"))
15    ;
16    camera->SetupAttachment(arm, USpringArmComponent::SocketName);
}
```

- **Tick():** aquí veremos que se realiza una llamada a la función auxiliar move(FVector pos1), la cual precisa de un argumento de tipo vector para funcionar. Esta realizará la función de mover el dron en el entorno seleccionado siguiendo el modelo de movimiento que vimos en el hito 1(10), solo que simplificado. En esta función también realizamos la llamada a la función de librería update(*vector3_transfer) que cuenta con un argumento de retorno en el que almacena el vector enviado por el publicador que luego se envía a la función move(). El argumento recogido por update es de un tipo definido previamente en el fichero de cabecera.

Listing 3.24: Definición del tipo vector3

```
1     struct vector3_transfer
2     {
3         float x, y, z;
4     };
5     typedef struct vector3_transfer vector3_transfer;
```

Listing 3.25: Función Tick()

```
1 void ADronePawnCom::Tick(float DeltaTime)
```

```

2 {
3     Super::Tick(DeltaTime);
4     //In here, we call the update function which will be constantly looking at
5     //the queue
6     //of published messages from the publisher in order to receive the
7     //coordiantes
8     update(&coordinates);
9     //We store the vector movement
10    pos.X=coordinates.x;
11    pos.Y=coordinates.y;
12    pos.Z=coordinates.z;
13    //And with the position vector created, we call the move function
14    move(pos);
15
16 }

```

Listing 3.26: Función Move()

```

1 void ADronePawnCom::move(FVector pos)
2 {
3     StartRelLocation = GetActorLocation();
4     //We compute the normalized movement
5     MoveOffsetNorm = pos;
6     MoveOffsetNorm.Normalize();
7     SetActorLocation(StartRelLocation + MoveOffsetNorm);
8 }

```

- **EndPlay()**: normalmente esta función únicamente hace un registro en la consola indicando que el peón ha sido eliminado del nivel y la razón. Sin embargo, esta función se puede sobrescribir, lo que quiere decir que como **BeginPlay()**, podemos crear una versión personalizada que será la que se ejecute. En nuestro caso la reescribimos y hacemos que ejecute la función **end()** de la librería, para finalmente liberar la memoria ocupada por la misma. Esto es necesario puesto que si no liberásemos la memoria cerrando la librería se provocaría un fallo en el programa si se tratase de ejecutar 2 veces seguidas.

Listing 3.27: Función EndPlay()

```

1 void ADronePawnCom::EndPlay(const EEndPlayReason::Type EndPlayReason)
2 {
3     Super::EndPlay(EndPlayReason);
4     UE_LOG(LogTemp, Warning, TEXT("Entrando en el end play"));
5     end();
6     dlclose(handle);
7     UE_LOG(LogTemp, Warning, TEXT("EndPlay completado"));
8 }

```

3.6.3. Resultados:

Como resultado de este hito podemos encontrar los ficheros adjuntados en el anexo en la página 48.

Además, se obtiene un peón el cual tiene la capacidad de moverse en un entorno de Unreal haciendo uso de unas coordenadas proporcionadas por un sistema de

comunicación publicador-suscriptor de ROS2. Sin embargo, aunque aquí hemos desarrollado el código fuente, es importante mencionar que para hacer este peón funcional es necesario realizar algunas configuraciones adicionales. Por ejemplo, será necesario crear un blueprint de esta clase de C++ y editarlo para añadir la malla del dron deseada y ajustar los ángulos del SpringArmComponent para orientar la cámara a donde se desee. Asimismo, al igual que en el hito 1 habremos de configurar el peón por defecto del juego base al blueprint creado mediante la misma línea solo que en este caso hemos de añadir un .Class al final

Listing 3.28: Configuracion del fichero GameModeBase.cpp

```
1 //Instanciamos que el default pawn sea el dron
2 static ConstructorHelpers::FClassFinder<APawn> DronePawnComBP(TEXT("/Game/
   Blueprints/DronePawnComBP"));
3 if(DronePawnComBP.Class != NULL)
4 {
5     DefaultPawnClass = DronePawnComBP.Class;
6 }
7 //DefaultPawnClass = ADronePawnCom::StaticClass();
```

3.7. Hito 6: Importación de escenarios a UE5

3.7.1. Descripción:

Este hito es similar al hito 2, en el que exploramos el funcionamiento de la importación de mallas. Sin embargo, en este caso nos centraremos en escenarios complejos con más de una malla y múltiples texturas.

3.7.2. Desarrollo:

Esta tarea es bastante sencilla en Unreal Engine tanto para escenarios importados de internet como para los creados específicamente. El proceso solo requiere de ficheros en formato .fbx y normalmente basta con arrastrar dicho fichero al cajón de contenidos del editor de Unreal, lo que provocará que aparezca una ventana de configuración. Esta ventana permite modificar varios campos de las mallas a importar. Normalmente no es necesario modificar nada y basta con importar el .fbx, guardar los contenidos importados, seleccionarlos y arrastrarlos al nivel en el que estemos trabajando.

Este método sirve cuando tenemos escenarios con pocas o ninguna textura o no requieran de cambios, es decir sean escenarios simples. Para este trabajo he decidido hacer uso de un mapa de Londres[Acc23], que cuenta con dos versiones, una texturizada y otra no.

La no texturizada sigue el método de importación descrito al principio, se selecciona el fichero, se arrastra, se guardan las mallas y ya está listo para ser añadido al nivel, generando el siguiente resultado



Figura 3.6.: Escenario de Londres sin texturas

Sin embargo, la texturizada necesita realizar algunas modificaciones. En el caso de este modelo 3D, de acuerdo a la guía proporcionada por su creador[Acc23b].

Lo primero es crear un material base en Unreal, para poder realizar configuraciones específicas de algunas de las mallas que vamos a importar, como por ejemplo el agua, a la que le daremos reflejos más adelante. Este material no tiene ningún requerimiento específico. Ahora, hemos de modificar este material y en su gráfico crear un “Texture Sample” dándole un nombre propio (BaseColor en este caso), convertirlo en un parámetro y configurar su textura base a cualquier textura.

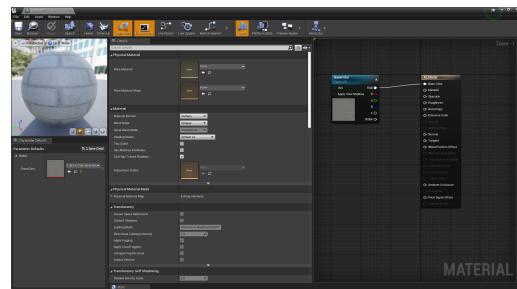


Figura 3.7.: Configuración inicial del material base

Esta versión texturizada del mapa de Londres, ha sido distribuida en 4 carpetas dado su gran tamaño. Estas representan cada una de las zonas de Londres, noreste, noroeste, sureste, suroeste, contando en su totalidad con casi 900 mallas. Las modificaciones que hemos de realizar en la ventana de importación se centran en su posición, pues hemos de modificar la x a x = -53199840 y la y a y = 18100590. Al mismo tiempo, hemos de hacer uso del material creado previamente; en la zona de MATERIAL hemos de seleccionar la opción de “crear nuevos materiales instanciados” en el campo “Material Import...” y seleccionar nuestro material con su textura base, la que creamos en su grafico (en mi caso BaseColor). Puesto que esta configuración es la misma para todos los ficheros basta con configurarlo una vez y esta se mantiene entre importaciones. Tras esto podemos importar todos los .fbx. Una vez tenemos todo en nuestro editor

3.7 Hito 6: Importación de escenarios a UE5

podemos seleccionarlos y arrastrarlos a la ventana del nivel para ver la ciudad de Londres a escala.

Una vez tenemos todas las mallas puestas en el mapa podemos añadir detalles al agua del río Támesis. Para esto hemos de modificar ese material base que creamos al principio. Hemos de añadirle un parámetro a su campo de "Roughness".

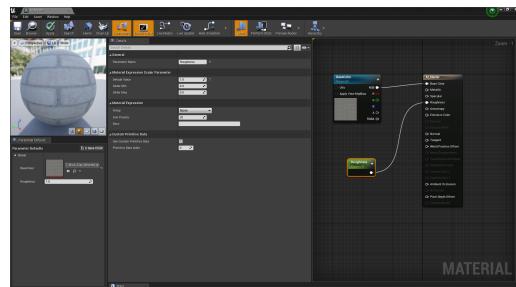


Figura 3.8.: Configuración final del material base

Una vez añadido y compilado, podemos entrar en las mallas del agua y cambiar este valor para darle reflejos y tener una textura más detallada.



Figura 3.9.: Escenario de Londres con texturas y reflejos en el agua

El proceso de importado de escenarios puede variar según el creador del escenario y cómo se configuren los ficheros a importar. Asimismo, el nivel de detalle de las mallas y texturas puede mejorar mucho dependiendo del creador de las mismas y de la potencia de la maquina que las use. En este campo Unreal Engine 5 tiene un gran potencial pudiendo llegar a tener una calidad fotorrealistica como puede ser ejemplo el juego UNRECORD[stu23b] que se encuentra en desarrollo por el estudio de videojuegos Drama[stu23a], que cuenta con visuales hiperrealistas, que asemejan a una grabación real.

3.7.3. Resultados:

Como se puede observar en las anteriores figuras el resultado es impresionante siendo esto un modelo gratuito y contando con un equipamiento que no es capaz de exprimir al máximo las capacidades gráficas de Unreal Engine 5.

El siguiente paso es añadir el dron al entorno y realizar pruebas de vuelo. Con esto estaría listo el visualizador de vuelo que sienta la base de un prototipo para el simulador.



Figura 3.10.: Dron dentro del mapa texturizado

3.8. Hito 7: Integración con Aerostack2 (en desarrollo)

3.8.1. Descripción:

Para realizar un visualizador más completo y que pueda replicar trayectorias reales de drones se busca añadir la herramienta de TF2 que nos permita recibir mensajes generados por Aerostack2.

4. Análisis de impacto

Las aplicaciones basadas en el avance de la tecnología y, sobre todo, los avances en el campo de los drones van a tener un impacto muy importante en la sociedad entre otras, facilitar el mantenimiento en plantas industriales, como las energéticas, su uso en cartografía, servicio postal y reparto a domicilio, incluso pudiendo ser usados con fines recreativos, entre otras muchas posibilidades.

Es por este amplio espectro de aplicaciones que los desarrollos como el realizado en este trabajo va a tener una gran presencia en la vida cotidiana, tanto en aspectos laborales como personales. Asociado a este desarrollo tecnológico será necesario disponer de entornos en los que realizar experimentos con drones de forma segura y que no involucren el uso de máquinas reales arriesgando su integridad. Es en este ámbito donde son necesarios los simuladores. Los entornos virtuales tridimensionales son capaces de simular casi a la perfección el comportamiento de un dron real dentro de un entorno generado por ordenador, permitiendo así hacer tantas pruebas como se desee de manera rápida, eficaz y segura optimizando así el proceso de producción. Desde el punto de vista medioambiental y de sostenibilidad los simuladores permitirán optimizar los procesos de desarrollo minimizando la necesidad de fabricaciones de prototipos y piezas de repuesto con la consiguiente reducción de material potencialmente contaminante.

Los elementos desarrollados a lo largo de este trabajo fundamentan una base sobre la que futuros desarrolladores podrán iterar para crear programas más avanzados que permitan simular al cien por ciento la realidad. Esta simulación está cada vez más cerca de nuestro alcance viendo los últimos avances en simulación de físicas, colisiones y capacidades gráficas de los entornos de desarrollo como Unreal Engine 5.

Las capacidades de estos robots y de este tipo de software solo están limitadas por nuestra imaginación, pudiendo llegar a entrenar drones en entornos virtuales gracias a programas de inteligencia artificial que luego podremos trasladar a máquinas reales. Esto por ejemplo se podría utilizar en la vigilancia y mantenimiento de bosques, para facilitar las tareas de los guardas forestales, por ejemplo minimizando el riesgo de incendios y por lo tanto ayudando así al mantenimiento de la vida de ecosistemas terrestres.

En conclusión, las herramientas presentadas, son una prueba de concepto de que es posible crear un sistema informático que nos ayude a mejorar la tecnología y el mundo que nos rodea.

Dentro de los siguientes pasos habría que contemplar la creación de objetos basados en la unión de componentes independientes que mejoren la modularidad

del dron y permitan gestionar más eficazmente la comunicación entre los componentes desde un actor propio. Para ello proponemos la creación de un peón e implementar en el mismo un control diferencial de la velocidad, prueba que describiremos más adelante.

5. Conclusiones

Como producto final de este trabajo se ha obtenido un visualizador de vuelo de un dron en el entorno de desarrollo de Unreal Engine 5, donde el manejo del mismo se realiza mediante nodos de ROS2.

El proyecto se ha desarrollado de forma líneal con las siguientes etapas intermedias:

1. Implementación del movimiento del dron en el visualizador.
2. Importación de un modelo tridimensional de un dron.
3. Implementación de un control diferencial de la velocidad. Que tendrá uso en futuras iteraciones.
4. Desarrollo de la infraestructura de comunicación basada en ROS2.
5. Integración de los resultados obtenidos en los apartados 1, 2, 3 y 4 en un peón de Unreal Engine 5.
6. Importación de entornos gráficos.

Este desarrollo constituye la base para el desarrollo final de un simulador mediante futuras iteraciones.

Desde mi perspectiva este trabajo ha sido una gran experiencia de aprendizaje, que me ha permitido investigar y desarrollar código utilizando lenguajes y entornos de gran interés. Así, he conseguido mis objetivos inicialmente planteados, que consistían en aprender un lenguaje que desconocía, C++, aprender a usar Unreal Engine 5 y conocer más a fondo el apasionante mundo de los drones.

A. Anexos

A.1. Código Hito 1:

Listing A.1: move.h

```
1 //Daniel Corrales 2023
2 //TFG UPM
3
4 #pragma once
5
6 #include "CoreMinimal.h"
7 #include "Components/SceneComponent.h"
8 #include "move.generated.h"
9
10 UDELEGATE(BlueprintAuthorityOnly)
11 DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FOnMoveReachEndPointSignature, bool,
12     IsTopEndpoint);
13
14 UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
15 class MOVETEST_API Umove : public USceneComponent
16 {
17     GENERATED_BODY()
18
19     public:
20         // Sets default values for this component's properties
21         Umove();
22
23         UFUNCTION(BlueprintCallable)
24         void EnableMovement(bool shouldMove);
25
26         UFUNCTION(BlueprintCallable)
27         void ResetMovement();
28
29         UFUNCTION(BlueprintCallable)
30         void SetMoveDirection(int Direction);
31
32     protected:
33         // Called when the game starts
34         virtual void BeginPlay() override;
35
36     public:
37         // Called every frame
38         virtual void TickComponent(float DeltaTime, ELevelTick TickType,
39             FActorComponentTickFunction* ThisTickFunction) override;
40
41     private:
42
43         //Offset to move
44         UPROPERTY(EditAnywhere)
45         FVector MoveOffset;
46
47         //Speed
48         UPROPERTY(EditAnywhere)
49         float Speed = 1.0f;
```

```

50     //Enable the movement of the component
51     UPROPERTY(EditAnywhere)
52     bool MoveEnable = true;
53
54     //On extreme reached event
55     UPROPERTY(BlueprintAssignable)
56     FOnMoveReachEndPointSignature OnEndpointReached;
57
58     //Computed locations
59     FVector StartRelativeLocation;
60     FVector MoveOffsetNorm;
61     float MaxDistance = 0.0f;
62     float CurDistance = 0.0f;
63     int MoveDirection = 1;
64
65 }
66 
```

Listing A.2: move.cpp

```

1 //Daniel Corrales 2023
2 //TFG UPM
3
4
5 #include "move.h"
6
7
8 // Sets default values for this component's properties
9 Umove::Umove()
10 {
11     // Set this component to be initialized when the game starts, and to be
12     // ticked every frame. You can turn these features
13     // off to improve performance if you don't need them.
14     PrimaryComponentTick.bCanEverTick = true;
15
16 }
17
18 void Umove::EnableMovement(bool shouldMove)
19 {
20     // Assign value and set correct tick enable state
21     MoveEnable = shouldMove;
22     SetComponentTickEnabled(MoveEnable);
23 }
24
25 void Umove::ResetMovement()
26 {
27     //Clear distance and set to origin
28     CurDistance = 0.0f;
29     SetRelativeLocation(StartRelativeLocation);
30 }
31
32 void Umove::SetMoveDirection(int Direction)
33 {
34     MoveDirection = Direction >= 1 ? 1 : -1;
35 }
36
37 // Called when the game starts
38 void Umove::BeginPlay()
39 {
40     Super::BeginPlay();
41
42     // Set start location
43     StartRelativeLocation = this->GetRelativeLocation();
44
45     //Compute normalized movement

```

A.1 Código Hito 1:

```
46     MoveOffsetNorm = MoveOffset;
47     MoveOffsetNorm.Normalize();
48     MaxDistance = MoveOffset.Size();
49
50     //Check if ticking is required
51     SetComponentTickEnabled(MoveEnable);
52 }
53
54
55
56 // Called every frame
57 void Umove::TickComponent(float DeltaTime, ELevelTick TickType,
58                           FActorComponentTickFunction* ThisTickFunction)
58 {
59     Super::TickComponent(DeltaTime, TickType, ThisTickFunction);
60
61     //Set the current distance
62     if(MoveEnable)
63     {
64         CurDistance += DeltaTime * Speed * MoveDirection;
65         if(CurDistance >= MaxDistance || CurDistance <= 0.0f){
66             //Inver direction
67             MoveDirection *= -1;
68
69             //Fire event
70             OnEndpointReached.Broadcast(CurDistance >= MaxDistance);
71
72             //Clamp distance
73             CurDistance = FMath::Clamp(CurDistance, 0.0f, MaxDistance);
74         }
75     }
76
77
78     // Compute and sert current location
79     SetRelativeLocation(StartRelativeLocation + MoveOffsetNorm * CurDistance);
80 }
```

Listing A.3: HoverComponent.h

```
1 //Daniel Corrales 2023
2 //TFG UPM
3
4 #pragma once
5
6 #include "CoreMinimal.h"
7 #include "Components/SceneComponent.h"
8 #include "HoverComponent.generated.h"
9
10
11 UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
12 class HOVERTEST_API UHoverComponent : public USceneComponent
13 {
14     GENERATED_BODY()
15
16 public:
17     // Sets default values for this component's properties
18     UHoverComponent();
19
20     //Funcion para poder setear el hover con una tecla
21     void InputHover(UInputComponent* PlayerInputComponent);
22
23     //Funcion para hacer true el hover
24     void SetHover();
25
26     //Funcion Hover
27     void Hover(float DeltaTime);
```

```

28         //Funcion para volver a la posicion de inicio
29         void ResetPosition();
30
31
32 protected:
33     // Called when the game starts
34     virtual void BeginPlay() override;
35
36 public:
37     // Called every frame
38     virtual void TickComponent(float DeltaTime, ELevelTick TickType,
39                               FActorComponentTickFunction* ThisTickFunction) override;
40
41 private:
42     //Toggle para saber si iniciar el hover o no
43     UPROPERTY(EditAnywhere)
44     bool TriggerHover = true;
45
46     //Toggle para saber si resetear el hover
47     UPROPERTY(EditAnywhere)
48     bool Reset = true;
49
50     //Altura en la que queremos el dron
51     UPROPERTY(EditAnywhere)
52     float Height = 500.0f;
53
54     //Velocidad a la que queremos que se mueva
55     UPROPERTY(EditAnywhere)
56     float Speed = 120.0f;
57
58     //Ubicaciones computadas o a computar
59     FVector StartRelativeLocation;
60     FVector HeightNorm;                                //Variable para
61     // normalizar el vector d ela posicion final y poder mover el dron de "
62     // poco en poco"
63     float MaxHeight = 0.0f;
64     float ActualHeight = 0.0f;
65 };

```

Listing A.4: HoverComponent.cpp

```

1 //Daniel Corrales 2023
2 //TFG UPM
3
4 #pragma once
5
6 #include "CoreMinimal.h"
7 #include "Components/SceneComponent.h"
8 #include "HoverComponent.generated.h"
9
10
11 UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
12 class HOVERTEST_API UHoverComponent : public USceneComponent
13 {
14     GENERATED_BODY()
15
16 public:
17     // Sets default values for this component's properties
18     UHoverComponent();
19
20     //Funcion para poder setear el hover con una tecla
21     void InputHover(UInputComponent* PlayerInputComponent);
22
23     //Funcion para hacer true el hover

```

A.2 Código Hito 3:

```
24     void SetHover();
25
26     //Funcion Hover
27     void Hover(float DeltaTime);
28
29     //Funcion para volver a la posicion de inicio
30     void ResetPosition();
31
32 protected:
33     // Called when the game starts
34     virtual void BeginPlay() override;
35
36 public:
37     // Called every frame
38     virtual void TickComponent(float DeltaTime, ELevelTick TickType,
39         FActorComponentTickFunction* ThisTickFunction) override;
40
41 private:
42     //Toggle para saber si iniciar el hover o no
43     UPROPERTY(EditAnywhere)
44     bool TriggerHover = true;
45
46     //Toggle para saber si resetear el hover
47     UPROPERTY(EditAnywhere)
48     bool Reset = true;
49
50     //Altura en la que queremos el dron
51     UPROPERTY(EditAnywhere)
52     float Height = 500.0f;
53
54     //Velocidad a la que queremos que se mueva
55     UPROPERTY(EditAnywhere)
56     float Speed = 120.0f;
57
58     //Ubicaciones computadas o a computar
59     FVector StartRelativeLocation;
60     FVector HeightNorm;                                //Variable para
61         "normalizar el vector de la posicion final y poder mover el dron de "
62         "poco en poco"
63     float MaxHeight = 0.0f;
64     float ActualHeight = 0.0f;
65
66 };
```

A.2. Código Hito 3:

Listing A.5: DronePawn.h

```
1 //Daniel Corrales 2023
2 //TFG UPM
3
4 #pragma once
5
6 #include "CoreMinimal.h"
7 #include "GameFramework/Pawn.h"
8 #include "DronePawn.generated.h"
9
10 UCLASS()
11 class ADronePawn : public APawn
12 {
13     GENERATED_BODY()
14 }
```

```

15 public:
16     // Sets default values for this pawn's properties
17     ADronePawn();
18
19     //Functions to modify the speed on each axis
20     void ModifySpeedX();
21     void ModifySpeedNegativeX();
22     void ModifySpeedY();
23     void ModifySpeedNegativeY();
24     void ModifySpeedZ();
25     void ModifySpeedNegativeZ();
26     void CalculateDelta();
27     void CalculateVelocity();
28
29 protected:
30     // Called when the game starts or when spawned
31     virtual void BeginPlay() override;
32
33 public:
34     // Called every frame
35     virtual void Tick(float DeltaTime) override;
36
37     // Called to bind functionality to input
38     virtual void SetupPlayerInputComponent(class UInputComponent*
39                                         PlayerInputComponent) override;
40
41 private:
42     //Variables for speed control
43     UPROPERTY(EditAnywhere)
44     FVector realVelocity;
45
46     UPROPERTY(EditAnywhere)
47     FVector deltaVelocity;
48
49     UPROPERTY(EditAnywhere)
50     FVector targetVelocity;
51
52     UPROPERTY(EditAnywhere)
53     float velocityConstant = 0.5f;
54
55     UPROPERTY(EditAnywhere)
56     float upperLimit = 1000.0f;
57
58     UPROPERTY(EditAnywhere)
59     float lowerLimit = -1000.0f;
60
61
62 };

```

Listing A.6: DronePawn.h

```

1 //Daniel Corrales 2023
2 //TFG UPM
3
4 #include "DronePawn.h"
5
6 // Sets default values
7 ADronePawn::ADronePawn()
8 {
9     // Set this pawn to call Tick() every frame. You can turn this off to
10     // improve performance if you don't need it.
11     PrimaryActorTick.bCanEverTick = true;
12 }
13

```

A.2 Código Hito 3:

```
14 // Called every frame
15 void ADronePawn::Tick(float DeltaTime)
16 {
17     Super::Tick(DeltaTime);
18
19     realVelocity = GetActorLocation() * DeltaTime;
20     CalculateDelta();
21     CalculateVelocity(); //CalculateVelocity
22 }
23
24 // Called when the game starts or when spawned
25 void ADronePawn::BeginPlay()
26 {
27     Super::BeginPlay();
28
29     targetVelocity = {0.0f, 0.0f, 0.0f};
30 }
31
32 // Called to bind functionality to input
33 void ADronePawn::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
34 {
35     Super::SetupPlayerInputComponent(PlayerInputComponent);
36
37     //Setting up bindings to add or subtract from the targetVelocity
38     PlayerInputComponent->BindAction("XPositive", IE_Pressed, this, &ADronePawn
39         ::ModifySpeedX);
40     PlayerInputComponent->BindAction("XNegative", IE_Pressed, this, &ADronePawn
41         ::ModifySpeedNegativeX);
42     PlayerInputComponent->BindAction("YPositive", IE_Pressed, this, &ADronePawn
43         ::ModifySpeedY);
44     PlayerInputComponent->BindAction("YNegative", IE_Pressed, this, &ADronePawn
45         ::ModifySpeedNegativeY);
46     PlayerInputComponent->BindAction("ZPositive", IE_Pressed, this, &ADronePawn
47         ::ModifySpeedZ);
48     PlayerInputComponent->BindAction("ZNegative", IE_Pressed, this, &ADronePawn
49         ::ModifySpeedNegativeZ);
50 }
51
52
53 //Functions to set the speed with keyboard presses
54 void ADronePawn::ModifySpeedX()
55 {
56     targetVelocity.X += 10.0f;
57 }
58 void ADronePawn::ModifySpeedNegativeX()
59 {
60     targetVelocity.X -= 10.0f;
61 }
62
63 void ADronePawn::ModifySpeedY()
64 {
65     targetVelocity.Y += 10.0f;
66 }
67
68 void ADronePawn::ModifySpeedNegativeY()
69 {
70     targetVelocity.Y -= 10.0f;
71 }
72
73 void ADronePawn::ModifySpeedZ()
74 {
75     targetVelocity.Z += 10.0f;
76 }
77
78 void ADronePawn::ModifySpeedNegativeZ()
```

```

75         targetVelocity.Z -= 10.0f;
76     }
77
78 /*
79 Function to calculate the change needed in the velocity in order to
80 get the drone to the target velocity
81 */
82 void ADronePawn::CalculateDelta()
83 {
84     deltaVelocity = (targetVelocity - realVelocity) * velocityConstant;
85 }
86
87 /*
88 Function to add to the realVelocity the values needed
89 to make the drone reach the intended velocity
90 */
91 void ADronePawn::CalculateVelocity()
92 {
93     if(deltaVelocity.X >= upperLimit)
94     {
95         deltaVelocity.X = upperLimit;
96     }
97     if(deltaVelocity.X <= lowerLimit)
98     {
99         deltaVelocity.X = lowerLimit;
100    }
101    if(deltaVelocity.Y >= upperLimit)
102    {
103        deltaVelocity.Y = upperLimit;
104    }
105    if(deltaVelocity.Y <= lowerLimit)
106    {
107        deltaVelocity.Y = lowerLimit;
108    }
109    if(deltaVelocity.Z >= upperLimit)
110    {
111        deltaVelocity.Z = upperLimit;
112    }
113    if(deltaVelocity.Z <= lowerLimit)
114    {
115        deltaVelocity.Z = lowerLimit;
116    }
117
118    realVelocity += deltaVelocity;
119 }

```

A.3. Código Hito 4:

Listing A.7: Fichero de cabecera del suscriptor ROS2

```

1 #ifdef __cplusplus
2 extern "C"
3 {
4 #endif
5
6     struct vector3_transfer
7     {
8         float x, y, z;
9     };
10    typedef struct vector3_transfer vector3_transfer;
11
12    void start();
13

```

A.3 Código Hito 4:

```
14     int update(vector3_transfer *);
15
16     int dummy(int num);
17
18     void end();
19 #ifdef __cplusplus
20 }
21#endif
```

Listing A.8: Fichero fuente del suscriptor ROS2

```
1 // Copyright 2016 Open Source Robotics Foundation, Inc.
2 //
3 // Licensed under the Apache License, Version 2.0 (the "License");
4 // you may not use this file except in compliance with the License.
5 // You may obtain a copy of the License at
6 //
7 //     http://www.apache.org/licenses/LICENSE-2.0
8 //
9 // Unless required by applicable law or agreed to in writing, software
10 // distributed under the License is distributed on an "AS IS" BASIS,
11 // WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 // See the License for the specific language governing permissions and
13 // limitations under the License.
14
15 #include <memory>
16 #include <deque>
17 #include <unistd.h>
18
19 #include "subscriber_member_function.h"
20
21 #include "rclcpp/rclcpp.hpp"
22 #include "geometry_msgs/msg/vector3.hpp"
23
24 using std::placeholders::_1;
25
26 class MinimalSubscriber : public rclcpp::Node
27 {
28 public:
29     MinimalSubscriber()
30         : Node("minimal_subscriber")
31     {
32         subscription_ = this->create_subscription<geometry_msgs::msg::Vector3>(
33             "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
34     }
35     std::deque<geometry_msgs::msg::Vector3::SharedPtr> deque;
36
37 private:
38     void topic_callback(const geometry_msgs::msg::Vector3::SharedPtr msg) // const
39     {
40         RCLCPP_INFO(this->get_logger(), "I heard: '%f' '%f' '%f'", msg->x, msg->y, msg
41             ->z);
42         deque.push_back(msg);
43     }
44     rclcpp::Subscription<geometry_msgs::msg::Vector3>::SharedPtr subscription_;
45 };
46
47 typedef rclcpp::executors::SingleThreadedExecutor Executor;
48
49 std::shared_ptr<Executor> executor;
50 //The node as a global variable
51 std::shared_ptr<MinimalSubscriber> node;
52
53 void start()
54 {
55     int argc = 1;
```

```

55     char name[] = "subscriber";
56     char *argv[] = {name};
57
58     rclcpp::init(argc, argv);
59     executor = std::make_shared<Executor>();
60     node = std::make_shared<MinimalSubscriber>();
61     executor->add_node(node);
62 }
63 // PILA
64 // OPCION 1: devolver void* en start, recibirlo en el update y hacer cast a
//             MinimalSubscriber
65 // OPCION 2: cola como atributo general, al principio del fichero
66
67 int update(vector3_transfer *ptr)
68 {
69     executor->spin_some();
70     if (!node->dequeue.empty())
71     {
72         auto msg = node->dequeue.front();
73         node->dequeue.pop_front();
74         ptr->x = msg->x;
75         ptr->y = msg->y;
76         ptr->z = msg->z;
77         return 1;
78     }
79     return 0;
80 }
81
82 void end()
83 {
84     node.reset();
85     rclcpp::shutdown();
86 }
87
88 int dummy(int num)
89 {
90     return num;
91 }
92
93 int main(int argc, char *argv[])
94 {
95     start();
96     vector3_transfer data;
97     for (int i = 0; i < 5; i++)
98     {
99         update(&data);
100        sleep(1);
101    }
102    end();
103    return 0;
104 }
```

Listing A.9: Fichero fuente del publicador ROS2

```

1 #include <chrono>
2 #include <memory>
3
4
5 #include "rclcpp/rclcpp.hpp"
6 #include "geometry_msgs/msg/vector3.hpp"
7
8 using namespace std::chrono_literals;
9
10 class CoordinatesPublisher : public rclcpp::Node
11 {
12
```

A.3 Código Hito 4:

```
13 public:
14
15     CoordinatesPublisher()
16     : Node("coordinates_publisher")
17     {
18         publisher_ = this->create_publisher<geometry_msgs::msg::Vector3>(
19             "topic", 10);
20         timer_ = this->create_wall_timer(
21             500ms, std::bind(&CoordinatesPublisher::timer_callback,
22                             this));
23     }
24
25 private:
26
27     void timer_callback()
28     {
29         auto message = geometry_msgs::msg::Vector3();
30
31         while(3 == scanf("%lf %lf %lf", &(message.x), &(message.y), &
32                         message.z))){//loop to read coordinates from file and send them
33             as a message
34             RCLCPP_INFO(this->get_logger(), "Publishing: '%f' '%f' '%f",
35                         message.x, message.y, message.z);
36             publisher_->publish(message);
37             sleep(5);
38         }
39     }
40
41
42
43 int main(int argc, char* argv[])
44 {
45
46     rclcpp::init(argc, argv);
47     rclcpp::spin(std::make_shared<CoordinatesPublisher>());
48     rclcpp::shutdown();
49     return 0;
50 }
```

Listing A.10: CMAKE del proyecto ROS2

```
1 cmake_minimum_required(VERSION 3.8)
2 project(droneCom)
3
4 if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
5     add_compile_options(-Wall -Wextra -Wpedantic)
6 endif()
7
8 # find dependencies
9 find_package(ament_cmake REQUIRED)
10 # uncomment the following section in order to fill in
11 # further dependencies manually.
12 # find_package(<dependency> REQUIRED)
13
14 find_package(ament_cmake REQUIRED)
15 find_package(rclcpp REQUIRED)
16 find_package(std_msgs REQUIRED)
17 find_package(geometry_msgs REQUIRED)
18
19 add_executable(talker src/CoordinatesPublisher.cpp)
20 ament_target_dependencies(talker rclcpp geometry_msgs)
```

```

21 add_executable(listener src/subscriber_member_function.cpp)
22 ament_target_dependencies(listener rclcpp geometry_msgs)
23
24 install(TARGETS
25   talker
26   listener
27   DESTINATION lib/${PROJECT_NAME})
28
29 add_library(listenerlib SHARED src/subscriber_member_function.cpp)
30 ament_target_dependencies(listenerlib rclcpp geometry_msgs)
31
32 if(BUILD_TESTING)
33   find_package(ament_lint_auto REQUIRED)
34   # the following line skips the linter which checks for copyrights
35   # comment the line when a copyright and license is added to all source files
36   set(ament_cmake_copyright_FOUND TRUE)
37   # the following line skips cpplint (only works in a git repo)
38   # comment the line when this package is in a git repo and when
39   # a copyright and license is added to all source files
40   set(ament_cmake_cpplint_FOUND TRUE)
41   ament_lint_auto_find_test_dependencies()
42 endif()
43
44 ament_package()

```

Listing A.11: package del proyecto ROS2

```

1 <?xml version="1.0"?>
2 <?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens
  ="http://www.w3.org/2001/XMLSchema"?>
3 <package format="3">
4   <name>droneCom</name>
5   <version>0.0.0</version>
6   <description>Package for C++ communication library for a drone simulation program
     in unreal</description>
7   <maintainer email="danielcorralesfalco@gmail.com">Daniel</maintainer>
8   <license>Apache License 2.0</license>
9
10  <buildtool_depend>ament_cmake</buildtool_depend>
11  <depend>rclcpp</depend>
12  <depend>std_msgs</depend>
13
14  <test_depend>ament_lint_auto</test_depend>
15  <test_depend>ament_lint_common</test_depend>
16
17  <export>
18    <build_type>ament_cmake</build_type>
19  </export>
20 </package>

```

A.4. Código Hito 5:**Listing A.12:** DronePawnCom.h

```

1 // Fill out your copyright notice in the Description page of Project Settings.
2
3 #pragma once
4
5 #include "CoreMinimal.h"
6 #include "GameFramework/Pawn.h"

```

A.4 Código Hito 5:

```
7 #include "DronePawnCom.generated.h"
8
9
10 UCLASS()
11 class ADronePawnCom : public APawn
12 {
13     GENERATED_BODY()
14
15 public:
16     // Constructor
17     ADronePawnCom();
18
19 protected:
20     // Called when the game starts or when spawned
21     virtual void BeginPlay() override;
22
23     //Called when the game is finished, modified to execute the end() function
24     //od the
25     //ROS library to shut down the listener. After that, the library is closed
26     virtual void EndPlay(const EEndPlayReason::Type EndPlayReason) override;
27
28     //Sets up components, called inside the constructor
29     void setComponents();
30
31     //Function to open the dynamic library and import its functions called in
32     //the BeginPlay
33     void dlImport();
34
35     //Function that moves the drone called in the Tick() function
36     void move(FVector pos1);
37
38     // Called every frame
39     virtual void Tick(float DeltaTime) override;
40
41     // Called to bind functionality to input
42     virtual void SetupPlayerInputComponent(class UInputComponent*
43         PlayerInputComponent) override;
44
45 private:
46     //Section 1: Here i'll define all the variables needed inside the pawn built to
47     //test the communication
48     void *handle;
49
50     //Start function, sets the listener up
51     typedef void (*fun_start)();
52     fun_start start;
53
54     //struct which holds the information needed in the update function
55     struct vector3_transfer
56     {
57         float x, y, z;
58     };
59     typedef struct vector3_transfer vector3_transfer;
60
61     //Update function will be called in the tick, used for receiving info from
62     //talker/publisher
63     typedef int (*fun_update)(vector3_transfer *);
64     fun_update update;
65
66     //End function for closing the library
67     typedef void (*fun_end)();
68     fun_end end;
69
70
71 //Section 2, here i'll declare the variables needed for moving the drone
```

```

69     //Arm component to have a static camera attached to it
70     UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category=Cam, meta=(
71         AllowPrivateAccess = "true"))
72     class USpringArmComponent* arm;
73
74     //To attach to the arm component and visualize the mesh
75     UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category=Cam, meta=(
76         AllowPrivateAccess = "true"))
77     class UCameraComponent* camera;
78
79     //Drone Mesh
80     UPROPERTY(EditAnywhere)
81     UStaticMeshComponent* MeshComponent;
82
83     //Starting location of the drone on each frame
84     FVector StartRelLocation;
85
86     //Normalized vector to set the movement each frame (for a fluid moving
87     //animation)
88     FVector MoveOffsetNorm;
89
90     //Coordinates which will be received by the subscriber and managed by the
91     //update function
92     vector3_transfer coordinates;
93
94     //Vector used to update the position of the pawn
95     UPROPERTY(EditAnywhere)
96     FVector pos;
97 }

```

Listing A.13: DronePawnCom.cpp

```

1 // Fill out your copyright notice in the Description page of Project Settings.
2
3
4 #include "DronePawnCom.h"
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <dlfcn.h>
8
9 #include "Camera/CameraComponent.h"
10 #include "GameFramework/SpringArmComponent.h"
11 // Sets default values
12 ADronePawnCom::ADronePawnCom()
13 {
14     // Set this pawn to call Tick() every frame. You can turn this off to
15     // improve performance if you don't need it.
16     PrimaryActorTick.bCanEverTick = true;
17     //We call the setComponents function to set up all the pawns components
18     setComponents();
19 }
20
21 // Called when the game starts or when spawned
22 void ADronePawnCom::BeginPlay()
23 {
24     Super::BeginPlay();
25     //We call the dlImport to open the ros2 library
26     dlImport();
27     //Here since unreal only executes this begin play once and at the start of the
28     //level
29     //we call the start function which sets up the listener in order to receive the
30     //coordinates from the ROS2 publisher/talker
31     start();

```

A.4 Código Hito 5:

```
31 }
32 }
33
34 // Called every frame
35 void ADronePawnCom::Tick(float DeltaTime)
36 {
37     Super::Tick(DeltaTime);
38     //In here, we call the update function which will be constantly looking at the
39     //queue
40     //of published messages from the publisher in order to recieve the coordiantes
41     update(&coordinates);
42     //We store the vector movement
43     pos.X=coordinates.x;
44     pos.Y=coordinates.y;
45     pos.Z=coordinates.z;
46     //And with the position vector created, we call the move function
47     move(pos);
48 }
49
50 // Called to bind functionality to input
51 void ADronePawnCom::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent
    )
52 {
53     Super::SetupPlayerInputComponent(PlayerInputComponent);
54 }
55 }
56
57 ////////////////////AUXILIARY FUNCTIONS///////////////////
58
59 void ADronePawnCom::EndPlay(const EEndPlayReason::Type EndPlayReason)
60 {
61     Super::EndPlay(EndPlayReason);
62     UE_LOG(LogTemp, Warning, TEXT("Entrando en el end play"));
63     end();
64     dlclose(handle);
65     UE_LOG(LogTemp, Warning, TEXT("EndPlay completado"));
66 }
67
68 void ADronePawnCom::setComponents()
69 {
70     //setup rootcomponent
71     RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("RootComponent"));
72     //Setup StaticMeshComponent
73     MeshComponent = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("DronMesh"));
74     MeshComponent->SetupAttachment(RootComponent);
75
76     //section for configuring the spring arm and camera attached to it
77     arm = CreateDefaultSubobject<USpringArmComponent>(TEXT("SpringArm"));
78     arm->SetupAttachment(RootComponent);
79
80     camera = CreateDefaultSubobject<UCameraComponent>(TEXT("CameraComponent"));
81     camera->SetupAttachment(arm, USpringArmComponent::SocketName);
82 }
83
84
85 void ADronePawnCom::dLIImport()
86 {
87     //Here i'll set the call to the dinamic library created for the ros2
88     //communication
89     //And all the functions neccesary in order to set up the functioning
90     //listener
91     handle = dlopen("/home/daniel/ros2_ws/build/droneCom/liblistenerlib.so",
92                     RTLD_LAZY);
93
94     if (!handle)
95     {
```

```

93     /* fail to load the library */
94     fprintf(stderr, "Error: %s\n", dlerror());
95     return;
96 }
97
98 //Once the library is open, we take the functions we want to use, in this case,
99 // start, update and end
100
101 void* temp1 = dlsym(handle, "start");
102 start = (fun_start)temp1;
103
104 if (!start)
105 {
106     /* no such symbol */
107     fprintf(stderr, "Error: %s\n", dlerror());
108     dlclose(handle);
109     UE_LOG(LogTemp, Error, TEXT("Error a la hora de cargar start"));
110     return;
111 }else{
112     UE_LOG(LogTemp, Warning, TEXT("Start importado"));
113 }
114
115 void* temp2 = dlsym(handle, "update");
116 update = (fun_update)temp2;
117
118 if (!update)
119 {
120     /* no such symbol */
121     fprintf(stderr, "Error: %s\n", dlerror());
122     dlclose(handle);
123     UE_LOG(LogTemp, Error, TEXT("Error a la hora de cargar update"));
124     return;
125 }else{
126     UE_LOG(LogTemp, Warning, TEXT("Update importado"));
127 }
128
129 void* temp3 = dlsym(handle, "end");
130 end = (fun_end)temp3;
131
132 if (!end)
133 {
134     /* no such symbol */
135     fprintf(stderr, "Error: %s\n", dlerror());
136     dlclose(handle);
137     UE_LOG(LogTemp, Error, TEXT("Error a la hora de cargar end"));
138     return;
139 }else{
140     UE_LOG(LogTemp, Warning, TEXT("End importado"));
141 }
142
143 void ADronePawnCom::move(FVector pos1)
144 {
145     StartRelLocation = GetActorLocation();
146     //We compute the normalized movement
147     MoveOffsetNorm = pos1;
148     MoveOffsetNorm.Normalize();
149     SetActorLocation(StartRelLocation + MoveOffsetNorm);
150 }
```

Listing A.14: droneSimGameModeBase.h

```

1 // Copyright Epic Games, Inc. All Rights Reserved.
2
3 #pragma once
4
```

A.4 Código Hito 5:

```
5 #include "CoreMinimal.h"
6 #include "DronePawn.h"
7 #include "GameFramework/GameModeBase.h"
8 #include "droneSimGameModeBase.generated.h"
9
10 /**
11  *
12  */
13 UCLASS()
14 class DRONESIM_API ADroneSimGameModeBase : public AGameModeBase
15 {
16     GENERATED_BODY()
17
18     //Creamos un constructor para que nos cargue el DronePawn como el default
19     // pawn del juego
20 public:
21     ADroneSimGameModeBase();
```

Listing A.15: droneSimGameModeBase.cpp

```
1 // Copyright Epic Games, Inc. All Rights Reserved.
2
3
4 #include "droneSimGameModeBase.h"
5 #include "DronePawn.h"
6 #include "DronePawnCom.h"
7
8
9 ADroneSimGameModeBase::ADroneSimGameModeBase()
10 {
11     //Instanciamos que el default pawn sea el dron
12     static ConstructorHelpers::FClassFinder<APawn> DronePawnComBP(TEXT("/Game/
13         Blueprints/DronePawnComBP"));
14     if(DronePawnComBP.Class != NULL)
15     {
16         DefaultPawnClass = DronePawnComBP.Class;
17     }
18     //DefaultPawnClass = ADronePawnCom::StaticClass();
19 }
```


Bibliografía

- [Acc23a] Accucities. Textured london model. *Accucities*, 2023, <https://www.accucities.com/portfolio-items/download-city-3d-london-autodesk-online-gallery/>.
- [Acc23b] Accucities. Textured london model import process. *Accucities*, 2023, <https://www.youtube.com/watch?v=W8u4Mi84Xss>.
- [ano22a] anon. Compite con los drones más veloces en el juego de carreras de drones, drl sim, ya disponible para descargar en epic gamesl. *epicgames*, 2022, <https://store.epicgames.com/es-ES/news/race-the-fastest-drones-in-the-drl-sim-drone-racing-game>.
- [ano22b] anon. Drones, la nueva herramienta imprescindible en seguridadl. *El blog de securitas*, 2022, <https://elblogdesecuritas.es/tecnologia/drones-herramienta-seguridad/l>.
- [Cgt23] Cgtrader. Cgtrader, 3d mesh web page. 2023, <https://www.cgtrader.com/free-3d-models?keywords=quadcopter>.
- [Eng23] Unreal Engine. Unreal engine documentation. *Unreal Engine forum*, 2004-2023, <https://docs.unrealengine.com/4.27/en-US/>.
- [Fus22a] Lötwig Fusel. Actors and components and custom move component | ue5 c++ tutorial. *Youtube*, 2022, <https://acortar.link/kn0diH>.
- [Fus22b] Lötwig Fusel. Editor module and component visualisation | ue5 c++ tutorial. *Youtube*, 2022, <https://acortar.link/rSCfsR>.
- [Mic17] Microsoft. Airsim announcement: This repository will be archived in the coming yearl. *microsoft github*, 2017, <https://microsoft.github.io/AirSim/>.
- [Mol19] Pilar Sánchez Molina. Tso optimiza y repotencia plantas solares con dronesl. *pv magazine*, 2019, <https://www.pv-magazine.es/2019/11/05/tso-optimiza-y-repotencia-plantas-solares-con-drones/>.
- [org23a] ROS2 organization. Ros2 tutorials: Begginer:client libraries. *ROS2 Humble documentation*, 2023, <https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries.html>.
- [org23b] ROS2 organization. Ros2 tutorials: Begginer:client libraries, creating a package. *ROS2 Humble documentation*,

- 2023, <https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Creating-Your-First-ROS2-Package.html>.
- [org23c] ROS2 organization. Ros2 tutorials: Begginer:client libraries, creating a workspace. *ROS2 Humble documentation*, 2023, <https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Creating-A-Workspace/Creating-A-Workspace.html>.
- [org23d] ROS2 organization. Ros2 tutorials: Begginer:client libraries, writing a simple publisher and subscriber (c++). *ROS2 Humble documentation*, 2023, <https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html>.
- [Rus21] Cristian Rus. Con 3.281 drones, este impresionante espectáculo en shanghai ha establecido un nuevo récord mundial. *xataka*, 2021, <https://www.xataka.com/drones/3-281-drones-este-impresionante-espectaculo-shanghai-ha-establecido-nuevo-record-mundial>.
- [stu23a] Drama studio. Drama studio page. 2023, <https://studiosdrama.com/>.
- [stu23b] Drama studio. Unrecord game. Steam, 2023, <https://store.steampowered.com/app/2381520/Unrecord/>.
- [Tap23a] Daniel Corrales ; Santiago Tapia. Tfg repository. *github*, 2023, <https://github.com/fjcorrales/TFG>.
- [Tap23b] Juan Pares ; Santiago Tapia. Tfg repository. *github*, 2023, <https://github.com/Pares-pg/TFG>.
- [yl21] slimeth ; Franco Fusco; Amaury Nègre; Huizerd; lukeopteran yun long. Flightmare repo. *github*, 2021, <https://github.com/uzh-rpg/flightmare>.