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

Aquí va el resumen del TFG. Extensión máxima 2 páginas.

Agradecimientos

Gracias

Índice general

1. Introducción

1.1. Descripción General

La simulación de drones es un campo relativamente nuevo, sin embargo, es un área en el que se han realizado numerosos desarrollos. En los recientes años se han realizado grandes avances e investigaciones con diversos fines, desde el estudio de la física del movimiento de los drones para su posterior simulación en entornos virtuales, hasta estudios para descubrir las posibles aplciaciones de estas versátiles máquinas. Como por ejemplo su uso en seguridad[2], mantenimiento de campos de placas solares[3] e incluso entretenimiento con enjambres de cientos de estos pequeños robots[1] .

Los drones son vehiculos 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 como mencionaba anteriormente. Es esta diversificación y popularización de los drones la que nos lleva a querer utilizarlos en problemas de la vida cotidiana. Por ello, simular el comportamiento de una máquina de este tipo se ha vuelto esencial, tanto para preveer que movimientos será capaz de realizar según en que condiciones se encuentre como para poder hacer pruebas sin el equipamiento real, evitando así posibles daños. Con esto en mente, se crean los simuladores, que sonentornos virtuales tridimensionales los cuales pueden recrear una gran variedad de situaciones.

Con respecto a los avances actuales en el campo de los simuladores, se han desarrollado una gran cantidad de los mismos, entre ellos, los que permiten aprender a controlar un dron, otros orientados a videojuegos de simulación de carreras de drones[4], etc. A nivel más profesional podemos encontrar algunos como AirSim[5], simulador de vuelo creado por Microsoft en el motor de juego de Unreal Engine (UE), o Flightmare[6], un simulador desarrollado para el motor de juego de Unity. Sin embargo, este simulador se basa en el simulador de físicas Gazebo, el cual para ciertas ocasiones es muy límitado.

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 el cual gestiona todo lo relacionado con la física que interactua 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 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 trayectoría del dron. La implementación de la gran mayoría del proyecto se realizará en C++. Al mismo tiempo, al crear este simulador en UE5, lo que se quiere es evitar el uso de Gazebo como motor de físicas ya que este es un tanto límitado. En cambio, se pretende simular la física y colisiones del dron con el propio motor de físicas de UE5.

1.2. Unreal Engine 5

Como he mencionado previamente, el simulador se va a desarrollar para el motor de juego Unreal Engine 5 (UE5) desarrollado por la compañía Epic Games. Esta herramienta es muy reciente y cuenta con unos avances gráficos enormes, pudiendo llegar a generar entornos que lucen casi idénticos a los reales dando la impresión de ser grabaciones del entorno natural y no simulaciones generadas por ordenador. Asimismo cuenta con un sistema de físicas y colisiones integrado, lo que fácilitará las tareas de implementación más adelante. Como añadido, este entorno también cuenta con herramientas de Inteligencia Artifical, que se pueden usar para controlar los actores, refiriendonos a los drones, que podamos 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, hacer el entrenamiento de los drones dentro del simulador para luego poder transferirlo a máquinas reales y así no arriesgarse a dañar el equipo.

1.3. ROS 2

ROS 2, o 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 se ha continuado desde entonces.

La versión de este framework con la que estamos trabajando es la versión Humble, la ultima versión publicada a la fecha de realización de este trabajo. Entrando a describir más específicamente en que 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 el paso de mensajes entre un controlador y la máquina en cuestión.

En este trabajo, el objetivo, además de poder tener un dron cuyo movimiento sea lo más fiel a la realidad posible, es implementar ROS 2 para poder realizar

el control del dron de forma “externa”, y así simular un vuelo real.

2. Desarrollo

2.1. 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 escribe en Unreal Engine 5 y ROS2. La fase de desarrollo se basa en 2 partes principales. Una primera de puesta a punto, o de prototipado, donde se ha experimentado con el motor de juego para descubrir la mejor forma de implementar los requisitos pedidos. Y una segunda fase en la que, ya , en esta primera versión se desarrolla la aplicación al completo.

2.2. Fase de prototipado

Esta ha sido la primera fase del desarrollo del simulador, en la cual he realizado algunas pruebas para encontrar la mejor forma de implementar un dron y su movimiento en el *sandbox* proporcionado por Unreal. Las pruebas realizadas han sido un total de 3 en relación a los elementos que personalmente he considerado más importantes para crear un simulador: el movimiento, tener una malla, o *mesh*, con la forma del dron y finalmente el control de la velocidad que usaremos para controlar la posición del dron.

2.2.1. Pruebas de movimiento de actores:

Para comenzar, se analizaron las posibilidades de movimiento de actores dentro de Unreal. En este aspecto cabe destacar que Unreal ofrece una serie clases propias que nos permiten crear elementos que situar en la escena a los cuales añadirles componentes. Estas clases son: actor, peón y personaje, donde cabe decir que las dos últimas heredan de actor, lo cual nos indica que todas las funciones de actor se encuentran disponibles tanto en peón y personaje. Estos elementos son los objetos que podemos añadir a un nivel o escena creado en Unreal, es decir son los elementos que interactuarán con el entorno y el usuario una vez comienza el juego. A pesar de que todas sean clases casi hermanas, las principales diferencias entre ellas son en relación a qué utilidad de Unreal son capaces de acceder. Los actores son los objetos más limitados ya que no ofrecen opción a que los maneje la IA ni permiten tener inputs de teclado. Los peones son el siguiente paso en la cadena, los cuales permiten ser controlados por la IA de Unreal y permiten inputs de teclado, sin embargo no tienen físicas integradas. Finalmente, los personajes son actores creados principalmente para

simular personas, estos incluyen todo lo mencionado previamente y además, integran las físicas necesarias para simular un humano.

En cuanto a los componentes que se pueden añadir a estos actores, son elementos que añaden funcionalidad a los mismos. Podemos encontrar 2 tipos de componentes: componentes de actor y componentes de escena. El componente de actor no hace nada esencialmente, sin embargo es imprescindible dado que es la base de la jerarquía del resto de componentes. Los componentes de escena son aquellos que podemos usar como apoyo para incluir funcionalidades a los actores.

Ahora, todos estos elementos se integran en el editor de Unreal mediante jerarquías donde predominan las acciones que tome el componente raíz de la misma.

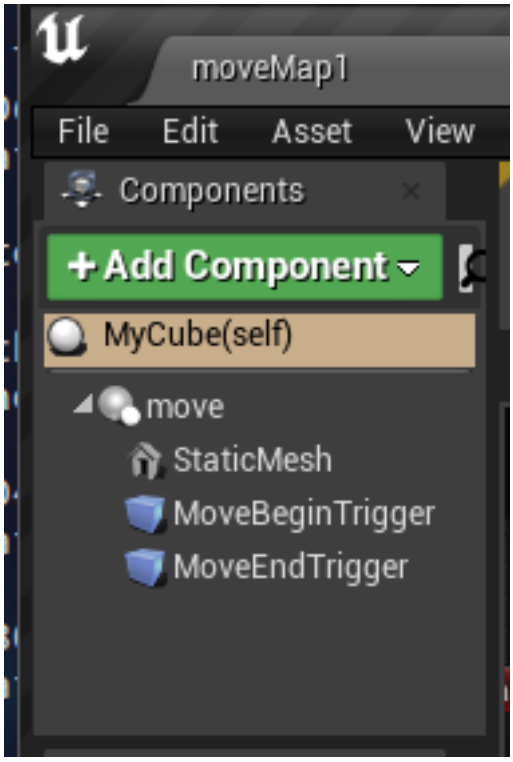

Figura 2.1.: Jerarquías de componentes en un blueprint

Haciendo uso de estos componentes diseñamos un primer prototipo 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[tutorial de youtube y página de Unreal] 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 3. En este caso, un cubo cuyo movimiento se basa en un vector de posición al cual se le podían ajustar las componentes tanto desde el editor como desde un *blueprint* como desde el propio código fuente.

A continuación abordamos la cuestión de las colisiones, añadiendo la propiedad

comunmente conocida como *Hitbox* (caja de colisiones). Para esto simplemente añadí un componente de escena de tipo hitbox, lo ajusté al tamaño de la mesh elegida y configuré un gráfico de eventos para que cuando el jugador entrara en contacto con el objeto este empezara a moverse. Asimismo incluí otra Hitbox para delimitar un área amplia alrededor de la malla para que en el caso de que detectara que el jugador abandonaba el área el objeto volviera a su posición original.

A continuación se adjuntan los ficheros “move.h” y “move.cpp” que contienen el código generado para este componente:

Listing 2.1: Header del componente de escena move

```
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,
       IsTopEndpoint);
12
13 UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
14 class MOVETEST_API Umove : public USceneComponent
15 {
16         GENERATED_BODY()
17
18 public:
19         // Sets default values for this component's properties
20         Umove();
21
22         UFUNCTION(BlueprintCallable)
23         void EnableMovement(bool shouldMove);
24
25         UFUNCTION(BlueprintCallable)
26         void ResetMovement();
27
28         UFUNCTION(BlueprintCallable)
29         void SetMoveDirection(int Direction);
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,
               FActorComponentTickFunction* ThisTickFunction) override;
39
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
51          //Enable the movement of the component
52          UPROPERTY(EditAnywhere)
53          bool MoveEnable = true;
54
55          //On extreme reached event
56          UPROPERTY(BlueprintAssignable)
57          FOnMoveReachEndPointSignature OnEndpointReached;
58
59          //Computed locations
60          FVector StartRelativeLocation;
61          FVector MoveOffsetNorm;
62          float MaxDistance = 0.0f;
63          float CurDistance = 0.0f;
64          int MoveDirection = 1;
65
66 };
```

Listing 2.2: Código fuente del componente de escena move

```
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
                ticked every frame.  You can turn these features
12          // off to improve performance if you don't need them.
13          PrimaryComponentTick.bCanEverTick = true;
14
15          // ...
16 }
17
18 void Umove::EnableMovement(bool shouldMove)
19 {
20          // Assing 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
```

```
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,
        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  }
```

Adjunto también el gráfico de eventos creado para completar su funcionalidad:

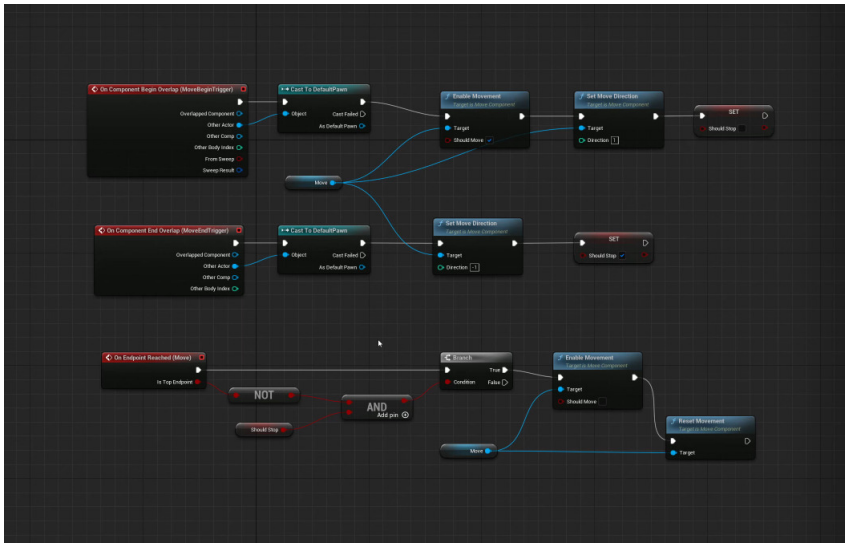

Figura 2.2.: Gráfico de eventos de activación de movimiento por colisión de hitbox

El código y el gráfico de eventos fue obtenido de la guía creada por Lötwig Fusel [referencia que toque]

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 modo 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. También se implementaron dos funciones, una para comenzar el vuelo y otra para resetearlo y cabe mencionar que esta vez no se ha implementado ningún grafico de eventos dado que la interacción de esta prueba se basa en mandar la señal de activación del hover o de reset.

Adjunto tanto el header como el código del componente en cuestión "*HoverComponent*":

Listing 2.3: Header del componente de escena HoverComponent

```
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
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,
                FActorComponentTickFunction* ThisTickFunction) override;
39
40  private:
41
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
                normalizar el vector d ela posicion final y poder mover el dron de "
                poco en poco"
61          float MaxHeight = 0.0f;
62          float ActualHeight = 0.0f;
63
64  };
```

Listing 2.4: Código fuente del componente de escena HoverComponent

```
1   //Daniel Corrales 2023
2   //TFG UPM
3
4
5   #include "HoverComponent.h"
6
7
8   // Sets default values for this component's properties
9   UHoverComponent::UHoverComponent()
10  {
11          // Set this component to be initialized when the game starts, and to be
                ticked every frame.  You can turn these features
12          // off to improve performance if you don't need them.
13          PrimaryComponentTick.bCanEverTick = true;
14
15          // ...
16  }
17
18  void UHoverComponent::InputHover(UInputComponent *PlayerInputComponent)
19  {
20          PlayerInputComponent->BindAction("Hover", IE_Pressed, this, &
                UHoverComponent::SetHover);
21  }
22
23  void UHoverComponent::SetHover()
24  {
25          TriggerHover = true;
26  }
27
28  void UHoverComponent::Hover(float DeltaTime)
29  {
30          //Establecemos la altura actual
31          ActualHeight += DeltaTime * Speed * 1;
```

```
32          //Computamos y actualizamos la ubicacion en caso de no haber llegado a la
                altura maxima
33          if(ActualHeight <= MaxHeight || ActualHeight <= 0.0f){
34                  SetRelativeLocation(StartRelativeLocation + HeightNorm *
                        ActualHeight);
35          }
36  }
37
38  void UHoverComponent::ResetPosition()
39  {
40          FVector start;
41          start.X = 0;
42          start.Y = 0;
43          start.Z = 40.0f;
44          SetRelativeLocation(start);
45  }
46
47
48  // Called when the game starts
49  void UHoverComponent::BeginPlay()
50  {
51          Super::BeginPlay();
52
53          //Obtenemos la ubicacion inicial del dron
54          StartRelativeLocation = GetRelativeLocation();
55
56
57          //Computamos el movimiento normalizado y la ubicacion final del dron
58          HeightNorm = StartRelativeLocation;
59          HeightNorm.Z += Height;
60          HeightNorm.Normalize();
61          MaxHeight = Height;
62  }
63
64  // Called every frame
65  void UHoverComponent::TickComponent(float DeltaTime, ELevelTick TickType,
        FActorComponentTickFunction* ThisTickFunction)
66  {
67          Super::TickComponent(DeltaTime, TickType, ThisTickFunction);
68
69          if(TriggerHover && !Reset)
70          {
71                  Hover(DeltaTime);
72          }
73
74          if(Reset && !TriggerHover)
75          {
76                  ResetPosition();
77          }
78  }
```

Tras desarrollar estos módulos de prueba basados en componentes de escena y observar su funcionamiento nuestro siguiente paso sería crear objetos basados en la unión de componentes independientes, lo que nos interesa en gran medida para proporcionarle cierta modularidad. La idea sería gestionar la comunicación entre los componentes desde un actor propio. Con este fin proponemos la creación de un peón e implementar en el mismo un control diferencial de la velocidad, prueba que describiremos más adelante.

2.2.2. Prueba de importación de mallas:

Durante el desarrollo de la prueba anteriormente descrita, busqué formas de poder integrar un modelo 3D, conocido como *mesh*, de un cuadricóptero en el editor de Unreal y como hacer que los componentes lo movieran.

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 pudiendo simplemente arrastrar el archivo al editor y tras una simple ventana de configuración tener ya disponible el modelo para su uso. El modelo seleccionado fue el siguiente:

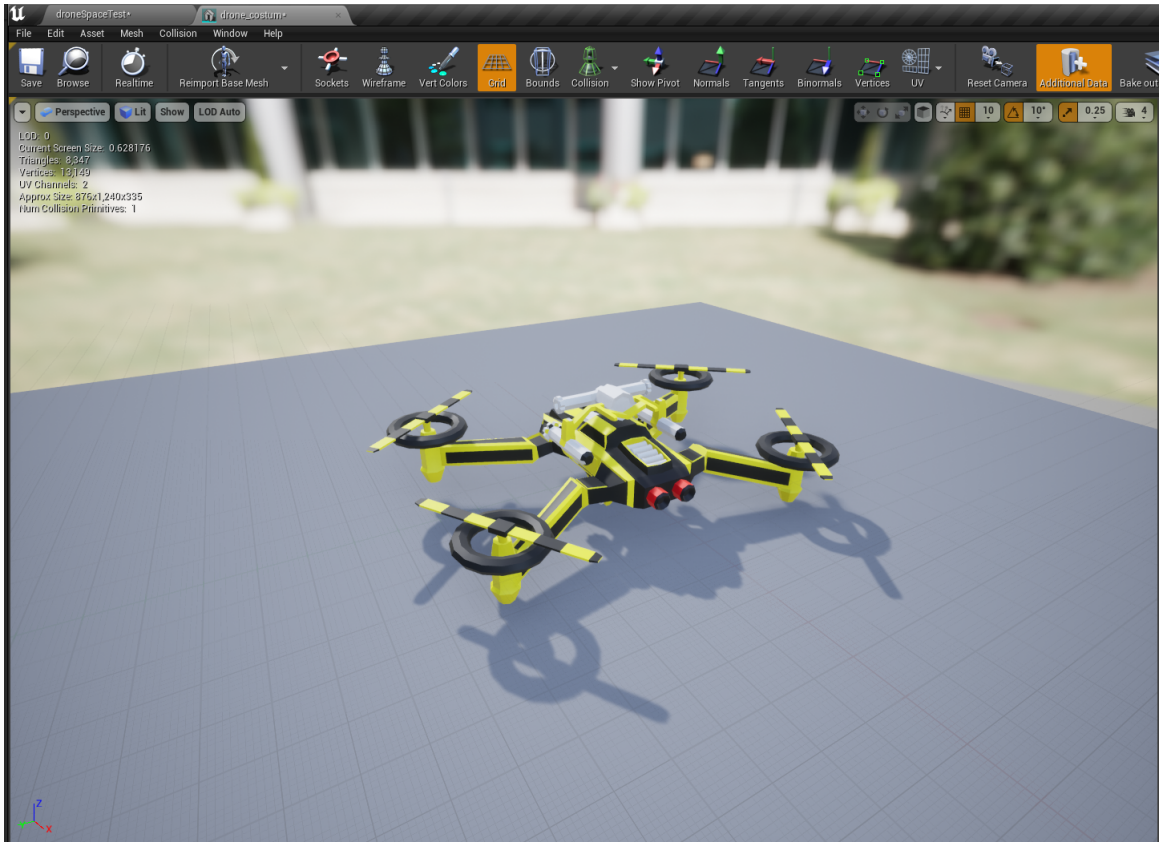

Figura 2.3.: Mesh del dron en el entorno de Unreal

Si se desea Epic Games también cuenta con una plataforma en la que los usuarios pueden subir sus modelos 3D ya sea de niveles u objetos a una tienda on-line. La desventaja es que la mayoría del contenido es de pago.

Con esto quedaba solventado el problema de poder tener un modelo tridimensional de un dron. LA siguiente cuestión será determinar si Unreal genera las colisiones automáticamente al proporcionarle una *hitbox* al modelo o si estas han de ser programadas desde 0.

2.2.3. Prueba de implementación de un control diferencial de la velocidad:

Finalmente y tras las primeras pruebas de movimiento decidimos crear un peón ya que este es el objeto ideal sobre el que crear un dron. Así pues, tras crear una clase en C++ de un peón, implementamos en la misma un control diferencial de la velocidad, un concepto simple basado en 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.

Es relevante considerar que Unreal proporciona una función *tick* dentro de todas sus clases, la cual se ejecuta a cada frame generado por el motor y ejecuta las instrucciones descritas en la misma. Al mismo tiempo, esta función nos proporciona una variable *DeltaTime* la cual lleva la cuenta del tiempo que pasa dentro del editor durante la ejecución del programa. Esta función será la que más carga computacional tenga al final, dado que es la que utilizaremos para poder generar un movimiento fluido en un futuro.

Otra función generada automáticamente y que merece la pena considerar es *BeginPlay*, esta se ejecuta en el primer instante en que se entra al nivel creado y solo esa vez. Esta resulta útil para establecer variables iniciales o si por ejemplo se desea almacenar la posición inicial de algún objeto.

Una vez explicados los conceptos anteriores pasamos a la implementación del control en cuestión, una implementación simple y que nos permite configurar una velocidad máxima, una mínima y una constante para ajustar la velocidad en el calculo del delta de la velocidad. La implementación queda tal que así:

Listing 2.5: Header del peón DronePawn

```
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
```

```
protected:
        // Called when the game starts or when spawned
        virtual void BeginPlay() override;

public:
        // Called every frame
        virtual void Tick(float DeltaTime) override;

        // Called to bind functionality to input
        virtual void SetupPlayerInputComponent(class UInputComponent*
            PlayerInputComponent) override;

private:

        //Variables for speed control
        UPROPERTY(EditAnywhere)
        FVector realVelocity;

        UPROPERTY(EditAnywhere)
        FVector deltaVelocity;

        UPROPERTY(EditAnywhere)
        FVector targetVelocity;

        UPROPERTY(EditAnywhere)
        float velocityConstant = 0.5f;

        UPROPERTY(EditAnywhere)
        float upperLimit = 1000.0f;

        UPROPERTY(EditAnywhere)
        float lowerLimit = -1000.0f;


};
```

Listing 2.6: Código fuente del peón DronePawn

```
//Daniel Corrales 2023
//TFG UPM

#include "DronePawn.h"

// Sets default values
ADronePawn::ADronePawn()
{
        // Set this pawn to call Tick() every frame.  You can turn this off to
            improve performance if you don't need it.
        PrimaryActorTick.bCanEverTick = true;

}

// Called every frame
void ADronePawn::Tick(float DeltaTime)
{
        Super::Tick(DeltaTime);

        realVelocity = GetActorLocation() * DeltaTime;
        CalculateDelta();
        CalculateVelocity(); //CalculateVelocity
        FVector pos = GetActorLocation();
        pos += DeltaTime*realVelocity;
        SetActorLocation(pos);
}

// Called when the game starts or when spawned
```

```
28 void ADronePawn::BeginPlay()
29 {
30         Super::BeginPlay();
31
32         targetVelocity = {0.0f, 0.0f, 0.0f};
33 }
34
35 // Called to bind functionality to input
36 void ADronePawn::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
37 {
38         Super::SetupPlayerInputComponent(PlayerInputComponent);
39
40         //Setting up bindings to add or subtract from the targetVelocity
41         PlayerInputComponent->BindAction("XPositive", IE_Pressed, this, &ADronePawn
               ::ModifySpeedX);
42         PlayerInputComponent->BindAction("XNegative", IE_Pressed, this, &ADronePawn
               ::ModifySpeedNegativeX);
43         PlayerInputComponent->BindAction("YPositive", IE_Pressed, this, &ADronePawn
               ::ModifySpeedY);
44         PlayerInputComponent->BindAction("YNegative", IE_Pressed, this, &ADronePawn
               ::ModifySpeedNegativeY);
45         PlayerInputComponent->BindAction("ZPositive", IE_Pressed, this, &ADronePawn
               ::ModifySpeedZ);
46         PlayerInputComponent->BindAction("ZNegative", IE_Pressed, this, &ADronePawn
               ::ModifySpeedNegativeZ);
47 }
48
49
50 //Functions to set the speed with keyboard presses
51 void ADronePawn::ModifySpeedX()
52 {
53         targetVelocity.X += 10.0f;
54 }
55
56 void ADronePawn::ModifySpeedNegativeX()
57 {
58         targetVelocity.X -= 10.0f;
59 }
60
61 void ADronePawn::ModifySpeedY()
62 {
63         targetVelocity.Y += 10.0f;
64 }
65
66 void ADronePawn::ModifySpeedNegativeY()
67 {
68         targetVelocity.Y -= 10.0f;
69 }
70
71 void ADronePawn::ModifySpeedZ()
72 {
73         targetVelocity.Z += 10.0f;
74 }
75
76 void ADronePawn::ModifySpeedNegativeZ()
77 {
78         targetVelocity.Z -= 10.0f;
79 }
80
81 /*
82 Function to calculate the change needed in the velocity in order to
83 get the drone to the target velocity
84 */
85 void ADronePawn::CalculateDelta()
86 {
87         deltaVelocity = (targetVelocity - realVelocity) * velocityConstant;
88 }
```

```
89
90 /*
91 Function to add to the realVelocity the values needed
92 to make the drone reach the intended velocity
93 */
94 void ADronePawn::CalculateVelocity()
95 {
96         if(deltaVelocity.X >= upperLimit)
97         {
98                 deltaVelocity.X = upperLimit;
99         }
100        if(deltaVelocity.X <= lowerLimit)
101        {
102                deltaVelocity.X = lowerLimit;
103        }
104        if(deltaVelocity.Y >= upperLimit)
105        {
106                deltaVelocity.Y = upperLimit;
107        }
108        if(deltaVelocity.Y <= lowerLimit)
109        {
110                deltaVelocity.Y = lowerLimit;
111        }
112        if(deltaVelocity.Z >= upperLimit)
113        {
114                deltaVelocity.Z = upperLimit;
115        }
116        if(deltaVelocity.Z <= lowerLimit)
117        {
118                deltaVelocity.Z = lowerLimit;
119        }
120
121        realVelocity += deltaVelocity;
122 }
```

Asimismo, gracias a utilizar un peón tenemos acceso a entradas de teclado mediante la función SetupPlayerInputComponent, la cual nos permite aumentar y disminuir los valores de las componentes del vector de la velocidad objetivo mediante teclas durante la ejecución del nivel. Muy relevante tener en cuenta que estos *keybinds* también han de configurarse dentro del editor de Unreal:

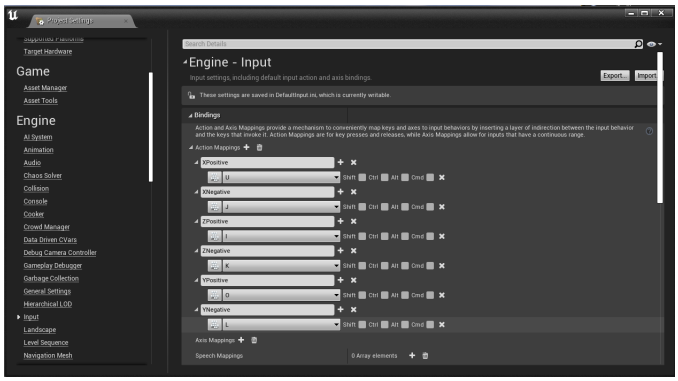

Figura 2.4.: Asociación de teclas en la configuración del proyecto

Y no solo eso, si no que además hemos de configurar el DefaultPawn del nivel al peón de la clase que hemos creado DronePawn, dado que es este "peón por

defecto" el que recoge las entradas del teclado y las transfiere al propio peón. Este cambio solamente lo podemos realizar desde el código del modo de juego que se generó automáticamente al crear el proyecto en Unreal. Dentro del archivo NombreProyectoGameModeBase.h y NombreProyectoGameModeBase.cpp hemos de añadir las 2 siguientes líneas (En mi caso el nombre de los archivos es droneSimGameModeBase.h y droneSimGameModeBase.cpp):

Listing 2.7: Header del modo de juego droneSimGameModeBase

```
1 // Copyright Epic Games, Inc. All Rights Reserved.
2
3 #pragma once
4
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
              pawn del juego
19 public:
20         AdroneSimGameModeBase();
21 };
```

Listing 2.8: Código fuente del modo de juego droneSimGameModeBase

```
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/
           Blueprints/DronePawnComBP"));
13     if(DronePawnComBP.Class != NULL)
14     {
15         DefaultPawnClass = DronePawnComBP.Class;
16     }
17     //DefaultPawnClass = ADronePawnCom::StaticClass();
18 }
```

Una vez realizados todos estos cambios veremos que aparece esto en el editor al ejecutar y seleccionar el peón:

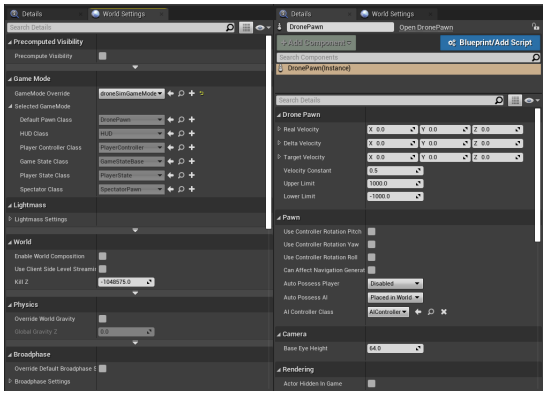

Figura 2.5.: Valores del editor del modo de juego y los vectores asociados al peón

Ya configurado y testado el correcto funcionamiento de las funciones previamente descritas, 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 realizar una integración con ROS 2 ya que pretendemos que el movimiento se rija mediante la velocidad que se enviará mediante los servicios de mensajería ofrecidos por ese framework.

2.3. Desarrollo final

Tras la realización de las pruebas anteriormente descritas, se realizo un reajuste de los objetivos planteados inicialmente. Finalmente, este trabajo consistirá en la realización de un visualizador de vuelo de un dron en un entorno tridimensional. Esto sentará las bases para futuros trabajos en los cuales se itere sobre los programas aquí desarrollados y completar así el simulador completo.

El desarrollo del visualizador de vuelo consistirá de tres partes: el desarrollo de un sistema publicador subscriptor de ROS2 para poder comunicar el dron dentro del entorno de UE5, el desarrollo del movimiento de un dron en base a las coordenadas publicadas en el publicador y la importación de escenarios tridimensionales en UE5 y sus capacidades.

2.3.1. Desarrollo de un sistema pub/sub de ROS2:

En la realidad, el control remoto de un dron es posible gracias a la biblioteca de C previamente descrita ROS2. Esta permite el intercambio de todo tipo de mensajes entre un publicador y un subscriptor, los cuales se pueden desarrollar libremente para cada caso particular. Para este visualizador de vuelo la intención es poder leer coordenadas de la entrada estandar, que el publicador lo convierta en un vector y lo publique para ser recibido posteriormente por el subscriptor. Este subscriptor va a ser un tanto especial puesto que ha de ser accesible desde un peón dentro de Unreal para poder moverlo más adelante.

Este último detalle mencionado es muy importante puesto que, para conseguirlo, habremos de hacer del subscriptor una librería que se pueda abrir en Unreal y permita usar sus funciones para recibir los mensajes del publicador. Esta tarea fue ampliamente simplificada gracias al trabajo de Juan Pares Guillen[link de su github] y Santiago Tapia, quienes desarrollaron un subscriptor 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 subscriptor 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 subscriptor y subscribirse 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 subscriptor y liberar la memoria ocupada.

Esta clase hace uso del tipo de mensaje Vector3, tipo propio de ROS2 el cual necesita que se incluya en la clase y posteriormente ha de indicarse en el CMAKE para su correcta compilación.

A continuación, en lo relacionado al publicador de ROS2 se toma como base el

publicador de ejemplo que ROS2 propone en sus tutoriales[ros2 tutoriales]. La base es la misma, solo que se hace uso de un tipo de mensaje diferente, el Vector3, igual que en el subscriptor. 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.

Con esto tenemos creados los ficheros fuente tanto del publicador como del subscriptor:

Listing 2.9: Header de subscriptor ueSub

```
#ifdef __cplusplus
extern "C"
{
#endif

    struct vector3_transfer
    {
        float x, y, z;
    };
    typedef struct vector3_transfer vector3_transfer;

    void start();

    int update(vector3_transfer *);

    int dummy(int num);

    void end();
#ifdef __cplusplus
}
#endif
```

Listing 2.10: Código fuente del subscriptor ueSub

```
// Copyright 2016 Open Source Robotics Foundation, Inc.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

#include <memory>
#include <deque>
#include <unistd.h>

#include "subscriber_member_function.h"

#include "rclcpp/rclcpp.hpp"
#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
         ->z);
41     deque.push_back(msg);
42   }
43   rclcpp::Subscription<geometry_msgs::msg::Vector3>::SharedPtr subscription_;
44 };
45
46 typedef rclcpp::executors::SingleThreadedExecutor Executor;
47
48 std::shared_ptr<Executor> executor;
49 //The node as a global variable
50 std::shared_ptr<MinimalSubscriber> node;
51
52 void start()
53 {
54   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->deque.empty())
71   {
72     auto msg = node->deque.front();
73     node->deque.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 2.11: Código fuente del publicador CoordinatesPublisher

```
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
13 public:
14
15         CoordinatesPublisher()
16         : Node("coordinates_publisher")
17         {
18                 publisher_ = this->create_publisher<geometry_msgs::msg::Vector3>("
                       topic", 10);
19                 timer_ = this->create_wall_timer(
20                         500ms, std::bind(&CoordinatesPublisher::timer_callback,
                               this));
21         }
22
23 private:
24
25         void timer_callback()
26         {
27                 auto message = geometry_msgs::msg::Vector3();
28
29                 while(3 == scanf("%lf %lf %lf", &(message.x), &(message.y), &(
                       message.z))){//loop to read coordinates from file and send them
                        as a message
30                         RCLCPP_INFO(this->get_logger(), "Publishing: '%f' '%f' '%f
                               '", message.x, message.y, message.z);
31                 publisher_->publish(message);
32                         sleep(5);
33                 }
34
35         }
36         rclcpp::TimerBase::SharedPtr timer_;
37         rclcpp::Publisher<geometry_msgs::msg::Vector3>::SharedPtr publisher_;
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 }
```

Ahora necesitamos poder compilarlos, para lo cual seguimos las indicaciones de la guía oficial de ROS2[guía de ROS2]. 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 indicar que se busque el paquete donde se encuentra el tipo de mensaje Vector3, el paquete "geometry_msgs", además de tener que usarlo también en los "add_executable" del publicador y del subscriptor. Finalmente es necesario añadir también un "add_library" para crear así la librería del subscriptor que queremos utilizar en Unreal.

Listing 2.12: CMAKE del proyecto

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

```
41   set(ament_cmake_cpplint_FOUND TRUE)
42   ament_lint_auto_find_test_dependencies()
43 endif()
44
45 ament_package()
```

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

El siguiente paso es gestionar el uso de la librería dentro de un peón en Unreal.

2.3.2. Desarrollo del peón en UE5:

Para este desarrollo final, decidimos hacer uso de un peón, por las razones mencionadas en la parte de prototipado. Este peón cumplirá dos funciones clave para este visualizador, la primera y más obvia gestionará la recepción de coordenadas mediante el uso de la librería de ROS2 previamente creada, la segunda función será mover el dron en el entorno de Unreal acorde con las coordenadas recibidas desde el publicador.

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 subscriptor. Como en la fase de prototipado, nos encontramos una vez más con las funciones BeginPlay(), Tick(), el constructor del propio peón, en este caso ADronePawnCom() y además aparecerá una función propia de Unreal no mencionada previamente, el EndPlay() solo que esta ha de ser sobrescrita y normalmente no se genera de forma automática en el fichero del peón. En cada una de estas podremos encontrar otras 3 funciones auxiliares las cuales implementan las funcionalidades deseadas. Al mismo tiempo se hará uso de las funciones importadas en 3 de las 4 funciones mencionadas anteriormente.

En la función BeginPlay(), veremos la función dlImport(), la cual se encarga de gestionar la apertura de la librería dinámica que contiene las funciones necesarias para hacer uso del subscriptor. 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”. Como se puede observar[insertar referencia] estamos importando estas funciones como punteros a función puesto que esta es la única manera de poder hacer uso de las mismas. Para que este método sea operativo es necesario definir en el header del archivo[insertar referencia] estas funciones mediante un typedef y crear aquí las variables en las que almacenaremos posteriormente estos punteros.

En el constructor podremos encontrar el método 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 y finalmente un RootComponent, del que colgar el resto de los componentes, y un MeshComponent para poder darle un cuerpo al dron.

A continuación, en la función de Tick(), veremos que se realiza una llamada a la función 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 la fase de prototipado en la página 8, solo que simplificado.

Finalmente, la función EndPlay(). Esta función es propia de todo actor de Unreal, y por herencia lo es de las clases peón y character. Esta se ejecuta al quitar, el peón del nivel caso que se da también al terminar el nivel o juego. De normal esta función unicamente hace un log 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 una función de la librería y finalmente libere la memoria ocupada por la misma. Esto es necesario puesto que si no liberásemos la memoria cerrando la librería esto provocaría que el programa fallase si se trata de ejecutar 2 veces seguidas.

Esto concluye la estructura general del dron. Ahora es importante mencionar el uso de la biblioteca dinámica. Las 3 funciones de las que se compone y las cuales utilizamos en este peón son: start(), update(vector3_transfer) y end(). La funcionalidad de cada una se puede ver en en la página 21. Sin embargo aquí vemos que la función update lleva un argumento, este es un argumento de retorno en el cual se almacenará el mensaje recibido para luego poder utilizar el vector en el movimiento del peón. Las llamadas a estas funciones se encuentran en lugares específicos dada la naturaleza de la ejecución en Unreal.

Vimos previamente en la fase de prototipado que las funciones se ejecutan de forma específica. El constructor y BeginPlay() se ejecutan un única vez al comenzar la ejecución general del nivel que contenga el peón, es por esto que aquí se realizan las llamadas a las funciones setComponents() y dllImport() para realizar la configuración inicial del dron. Además de esas dos funciones aquí ya realizamos la primera llamada a una de las funciones de la librería, la función start() para inicializar el subscriptor. Prestando atención a la función de Tick(), esta se ejecuta en cada fotograma procesado por Unreal, por lo que en esta realizamos la segunda llamada funciones de librería, en este caso a update(), para recibir los mensajes con los vectores de posición a donde deseamos mover el dron. Es por este comportamiento de ”bucle” por lo que aquí también llamamos a la función move(). Finalmente la función EndPlay() se ejecuta una sola vez al terminar la ejecución y aquí realizamos la tercera y última llamada a la librería con la función end() que se encarga de cerrar el subscriptor.

Con este desarrollo obtenemos el código fuente del peón “DronePawnCom”

Listing 2.13: Header de 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"
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
               od the
24         //ROS library to shut down the listener. After that, the library is closed
25         virtual void EndPlay(const EEndPlayReason::Type EndPlayReason) override;
26
27         //Sets up components, called inside the constructor
28         void setComponents();
29
30         //Function to open the dynamic library and import its functions called in
               the BeginPlay
31         void dlImport();
32
33         //Function that moves the drone called in the Tick() function
34         void move(FVector pos1);
35
36 public:
37         // Called every frame
38         virtual void Tick(float DeltaTime) override;
39
40         // Called to bind functionality to input
41         virtual void SetupPlayerInputComponent(class UInputComponent*
               PlayerInputComponent) override;
42
43 private:
44 //Section 1: Here i'll define all the variables needed inside the pawn built to
       test the communication
45         void *handle;
46
47         //Start function, sets the listener up
48         typedef void (*fun_start)();
49         fun_start start;
50
51         //struct which holds the information needed in the update function
52          struct vector3_transfer
53     {
54         float x, y, z;
55     };
56         typedef struct vector3_transfer vector3_transfer;
57
58         //Update function will be called in the tick, used for recieving info from
               talker/publisher
59         typedef int (*fun_update)(vector3_transfer *);
60         fun_update update;
61
62         //End function for closing the library
63         typedef void (*fun_end)();
64         fun_end end;
```

```
65
66
67
68 //Section 2, here i'll declare the variables needed for moving the drone
69
70         //Arm component to have a static camera attached to it
71         UPROPERTY(VisibleAnywhere,BlueprintReadOnly,Category=Cam,meta=(
               AllowPrivateAccess = "true"))
72         class USpringArmComponent* arm;
73
74         //To attarch to the arm component and visualize the mesh
75         UPROPERTY(VisibleAnywhere,BlueprintReadOnly,Category=Cam,meta=(
               AllowPrivateAccess = "true"))
76         class UCameraComponent* camera;
77
78         //Drone Mesh
79         UPROPERTY(EditAnywhere)
80     UStaticMeshComponent* MeshComponent;
81
82         //Starting location of the drone on each frame
83         FVector StartRelLocation;
84
85         //Normalized vector to set the movement each frame (for a fluid moving
               animation)
86         FVector MoveOffsetNorm;
87
88         //Coordinates wich will be recieved by the subscriptor and managed by the
               update function
89         vector3_transfer coordinates;
90
91         //Vector used to update the position of the pawn
92         UPROPERTY(EditAnywhere)
93         FVector pos;
94
95 };
```

Listing 2.14: Código fuente de DronePawnCom

```
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
               improve performance if you don't need it.
15         PrimaryActorTick.bCanEverTick = true;
16     //We call the setComponents function to set up all the pawns components
17     setComponents();
18 }
19
20 // Called when the game starts or when spawned
21 void ADronePawnCom::BeginPlay()
22 {
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
```

```
        level
28      //we call the start function which sets up the listener in order to recieve the
29      //coordinates from the ROS2 publisher/talker
30      start();
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
            queue
39      //of published messages from the publisher in order to recieve the coordiantes
40      update(&coordinates);
41          //We store the vector mevement
42      pos.X=coordinates.x;
43      pos.Y=coordinates.y;
44      pos.Z=coordinates.z;
45      //And with the postition vector created, we call the move function
46      move(pos);
47
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
77      //section for configuring the spring arm and camera attached to it
78      arm = CreateDefaultSubobject<USpringArmComponent>(TEXT("SpringArm"));
79      arm->SetupAttachment(RootComponent);
80
81      camera = CreateDefaultSubobject<UCameraComponent>(TEXT("CameraComponent"));
82      camera->SetupAttachment(arm,USpringArmComponent::SocketName);
83  }
84
85  void ADronePawnCom::dlImport()
86  {
87      //Here i'll set the call to the dinamic library created for the ros2
            comunication
88          //And all the functions neccesary in order to set up the functioning
                listener
89          handle = dlopen("/home/daniel/ros2_ws/build/droneCom/liblistenerlib.so",
```

```
            RTLD_LAZY);

        if (!handle)
    {
        /* fail to load the library */
        fprintf(stderr, "Error: %s\n", dlerror());
        return;
    }

    //Once the library is open, we take the finctions we want to use, in this case,
         start, update and end

    void* temp1 = dlsym(handle, "start");
    start = (fun_start)temp1;

    if (!start)
    {
        /* no such symbol */
        fprintf(stderr, "Error: %s\n", dlerror());
        dlclose(handle);
        UE_LOG(LogTemp, Error, TEXT("Error a la hora de cargar start"));
        return;
    }else{
        UE_LOG(LogTemp, Warning, TEXT("Start importado"));
    }

    void* temp2 = dlsym(handle, "update");
    update = (fun_update)temp2;

    if (!update)
    {
        /* no such symbol */
        fprintf(stderr, "Error: %s\n", dlerror());
        dlclose(handle);
        UE_LOG(LogTemp, Error, TEXT("Error a la hora de cargar update"));
        return;
    }else{
        UE_LOG(LogTemp, Warning, TEXT("Update importado"));
    }

    void* temp3 = dlsym(handle, "end");
    end = (fun_end)temp3;

    if (!end)
    {
        /* no such symbol */
        fprintf(stderr, "Error: %s\n", dlerror());
        dlclose(handle);
        UE_LOG(LogTemp, Error, TEXT("Error a la hora de cargar end"));
        return;
    }else{
        UE_LOG(LogTemp, Warning, TEXT("End importado"));
    }
}

void ADronePawnCom::move(FVector pos1)
{
    StartRelLocation = GetActorLocation();
        //We compute the normalized movement
        MoveOffsetNorm = pos1;
        MoveOffsetNorm.Normalize();
    SetActorLocation(StartRelLocation + MoveOffsetNorm);
}
```

2.3.3. Importación de escenarios a UE5:

Uno de los objetivos de este trabajo es mantener una modularidad entre el entorno y el dron en si. Esto es bastante sencillo 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 y esto provocará que aparezca una ventana de configuración. Esta ventana permite modificar varios campos de las mallas a importar. De normal 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 mallas simples. Para este trabajo he decidido hacer uso de un mapa de Londres[referencia a la página del modelo]. Este modelo 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

(fotografías del entorno)

Sin embargo, la texturizada necesita realizar algunas modificaciones. En el caso de este modelo 3D, de acuerdo a la guía proporcionada por su creador[referencia a la guía]. En esta se hablan de los pasos a seguir para la correcta importación del mapa con sus texturas.

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, convertirlo en un parámetro y configurar su textura base a cualquier textura.

(insertar foto de la configuración del material)

Esta versión texturizada del mapa de Londres, ha sido distribuida en 4 carpetas dado su gran tamaño. Estas 4 representan cada una una zona 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 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 rio Támesis. Para esto hemos de modificar ese material base que creamos al principio. Hemos de añadirle un parámetro a su campo de ”Roughness”.

(foto del grafico)

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

(foto del agua antes y después)

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[referencia al juego] que se encuentra en desarrollo por el estudio de videojuegos *Drama*[Referencia al estudio], que cuenta con visuales hiperrealistas, que asemejan a una grabación real.

3. Resultados

Con este desarrollo hemos conseguido crear un sistema que permite visualizar el movimiento de un dron en un entorno tridimensional de acuerdo a las coordenadas proporcionadas.

En este caso la modularidad es completa, lo que tiene la ventaja de que todos los elementos se pueden configurar de manera independiente, ajustandose de manera flexible a los diseños del usuario.

Para ilustrar la operatividad de la herramienta desarrollada se describe un caso de uso de los elementos creados.

3.1. Caso de uso

Como resultado del desarrollo final de este trabajo se ha producido una considerable cantidad de código: el código fuente de un peón de Unreal Engine y un publicador y subscriptor de ROS2 así como una librería del subscriptor. Dado que uno de los objetivos de este proyecto era conseguir una modularidad entre el objeto del dron (con su gestión del paso de mensajes y movimiento en el entorno) y el entorno, el dron se proporciona mediante su código fuente y requiere de una configuración específica de los ficheros fuente para poder hacer uso del mismo en un escenario. Esto nos permite poder mover el dron de proyecto en proyecto sin limitaciones, proporcionando total libertad para elegir el escenario.

La forma en la que un usuario debería de configurar el entorno para ejecutar el visualizador es la siguiente: lo primero es crear una clase de C++ de tipo peón en el proyecto de Unreal y darle el nombre que se desee (el nombre de la clase proporcionada es DronePawnCom, por lo que si el usuario elije uno distinto habrá de cambiarlo en su clase). Una vez esté creado ha de copiar el código que contienen los ficheros del dron (DronePawnCom.cpp y DronePawnCom.h) en sus respectivos ficheros dentro de su proyecto. Con esto ya tiene el código fuente del dron importado. El motivo de realizarlo de esta manera y no proporcionar lo comunmente conocido dentro de Unreal como “Blueprint” es para tener acceso directo al código fuente del peón. Ahora solo necesita crear un blueprint de esa clase de C++, para esto basta con hacer click derecho en la clase de C++ dentro de su carpeta de contenidos del editor y verá la opción de “create blueprint”. Finalmente necesita hacer un ajuste en el modo de juego, esta es una clase la cual contiene ajustes del “mundo” o nivel de Unreal que va a ejecutar. Aquí ha de establecer como peón predeterminado el blueprint que ha creado previamente, para ello ha de crear un constructor en el header y dentro del fichero

fuente (el .cpp) habrá de crear la función y añadir la siguiente línea de código "DefaultPawnClass = nombreDeSuBlueprint.Class;".

(foto de la línea del gamemodeBase)

Tras esto, si también tiene configurado el paquete de ROS2 con el publicador y la librería del subscriptor (como se mencionaba en la página en la página 21), solo tiene que modificar en la clase del peón el path donde se encuentre la librería en la función dlopen() para configurar correctamente la apertura de la biblioteca dinámica.

(foto de la línea del dlopen)

Una vez configurado todo solo falta establecer la malla del dron. Para esto basta con entrar en el editor del blueprint creado previamente, seleccionar el componente de mesh en la jerarquía y seleccionar la malla que hayamos importado al proyecto que deseemos utilizar.

(foto de la configuración de la malla)

Si todo esto se realizó correctamente, solo falta compilar y el proyecto ya estaría listo para ejecutarse. La ejecución es muy simple, nada más tener todo compilado, basta con iniciar el nivel; inicialmente el dron no se moverá, pues está esperando a recibir mensajes del publicador. Para poder movelo, es necesario ejecutar el publicador transfiriendole por la entrada estándar los contenidos de un fichero que contenga coordenadas. El formato de estas ha de ser de 3 enteros separados por espacios y un salto de línea entre cada set de coordenadas. Como ejemplo está el fichero "coordinates.txt" que hace un recorrido simple.

Listing 3.1: Fichero de coordenadas de ejemplo

```
1 0 0 100
2 100 0 0
3 0 100 0
4 0 -100 0
5 -100 0 0
6 0 0 -100
7 0 0 0
```

Con esto se obtiene modularidad de la que se hablaba anteriormente, pues el dron y el escenaraio no comparten nada, siendo totalmente independientes uno del otro.

4. Análisis de impacto

El impacto de este trabajo se deriva en su potencial en el desarrollo de software aplicado a simuladores a futuro. 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. Desde el punto de vista medioambiental y de desarrollo sostenible los simuladores permitirán optimizar los procesos de desarrollo minimizando la necesidad de fabricaciones de prototipos y piezas de repuesto.

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 traspasar 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.

A. Anexos

A.1. Overview

Bibliografía

[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.

[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/.

[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.

[yl21] slimeth ; Franco Fusco; Amaury Nègre; Huizerd; lukeopteran yun long. Flightmare repo. *github*, 2021, https://github.com/uzh-rpg/flightmare.