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

Índice general

1 Introducción **1**
1.1 Descripción General 1
1.2 Unreal Engine 5 2
1.3 ROS 2 2

2 Desarrollo **5**
2.1 Descripción General 5
2.2 Fase de prototipado 5
2.2.1 Pruebas de movimiento de actores: 5
2.2.2 Prueba de importación de mallas: 13
2.2.3 Prueba de implementación de un control diferencial de la velocidad: 14

Bibliografía **21**

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 límitados 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 componenetes de escena. El componente de actor no hace nada esencialmente, sin embargo es imprescindible dado que es la base de la jeraquía 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.

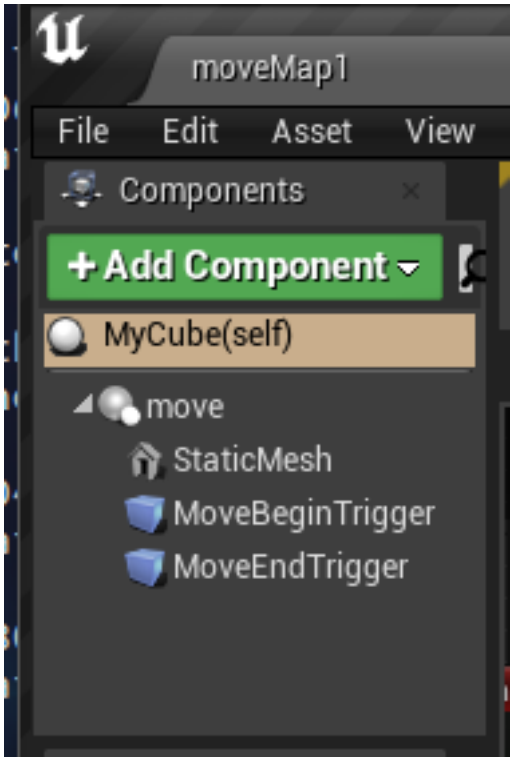

Figura 2.1: Jerarquías de componentes en un blueprint

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:

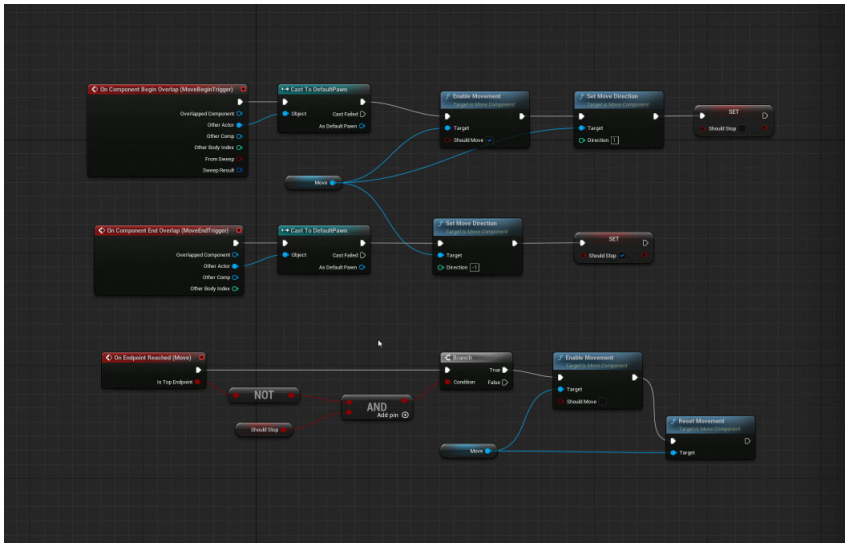

Figura 2.2: Gráfico de eventos de activación de movimiento por colisión de hitbox

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 comunición entre los componentes desde un actor propio. Con este fin proponemos la creación de un peon 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:

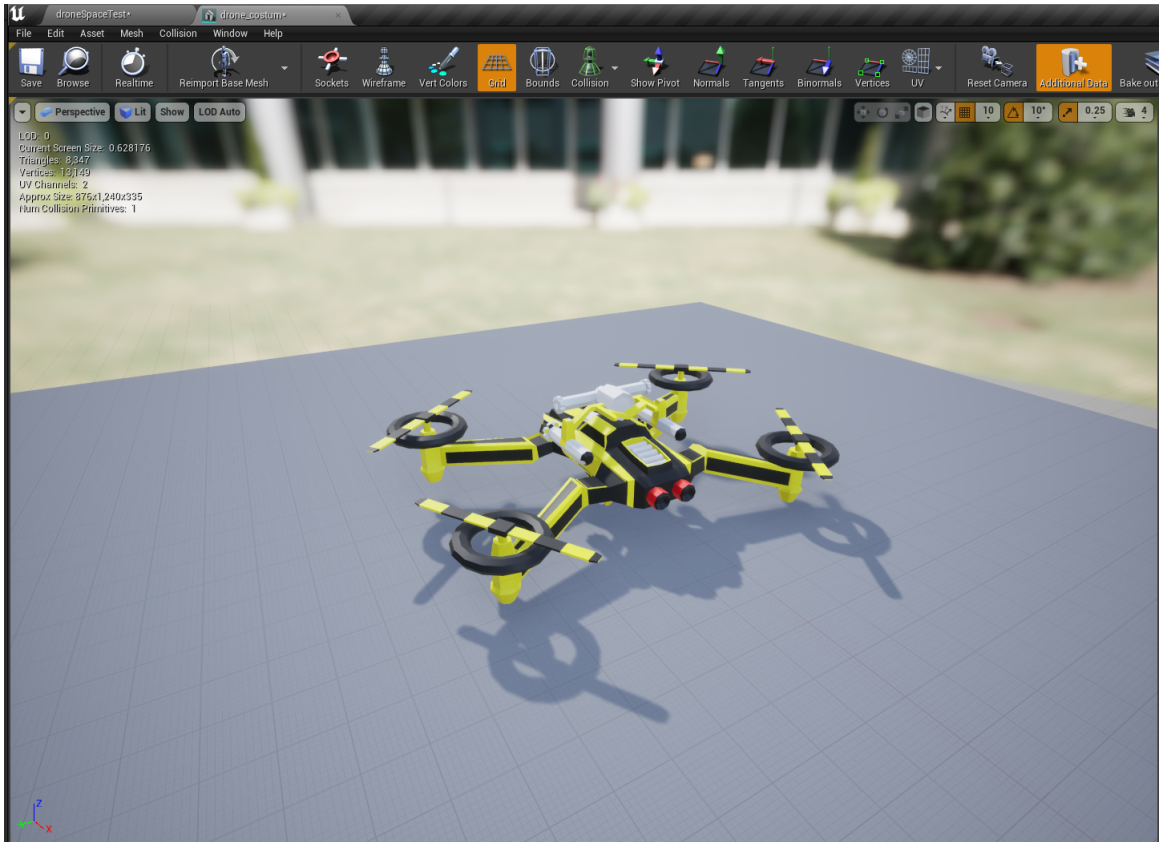

Figura 2.3: Mesh del dron en el entorno de Unreal

Si se desea Epic Games también cuenta con una paltaforma en la que los usuarios pueden subir sus modelos 3D ya sea de niveles u objetos a una tienda online. 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 objeo 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 poporciona 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 utíl 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 minima 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 DRONESIM_API 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:

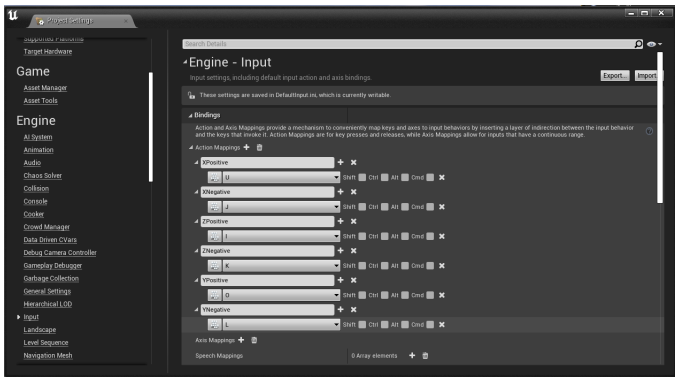

Figura 2.4: Asociación de teclas en la configuración del proyecto

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 recoje 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 autómaticamente 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 //Daniel Corrales 2023
3 //TFG UPM
4
5 #pragma once
6
7 #include "CoreMinimal.h"
8 #include "DronePawn.h"
9 #include "GameFramework/GameModeBase.h"
10 #include "droneSimGameModeBase.generated.h"
11
12 /**
13  *
14  */
15 UCLASS()
16 class DRONESIM_API AdroneSimGameModeBase : public AGameModeBase
17 {
18         GENERATED_BODY()
19
20         //Creamos un constructor para que nos cargue el dronePawn como el default
              pawn del juego
21 public:
22         AdroneSimGameModeBase();
23 };
```

Listing 2.8: Código fuente del modo de juego droneSimGameModeBase

```
1 // Copyright Epic Games, Inc. All Rights Reserved.
2 //Daniel Corrales 2023
3 //TFG UPM
4
5
6 #include "droneSimGameModeBase.h"
7 #include "DronePawn.h"
8
9 AdroneSimGameModeBase::AdroneSimGameModeBase()
10 {
11     //Instanciamos que el default pawn sea el dron
12     DefaultPawnClass = ADronePawn::StaticClass();
13 }
```

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

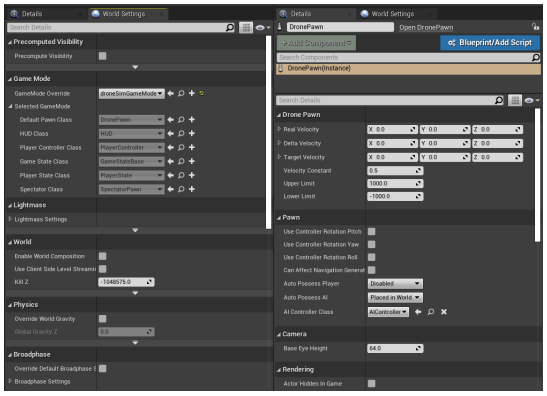

Figura 2.5: Valores del editor del modo de juego y los vectores asociados al peón

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 concluímos 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.

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.