ESCUELA DE ARQUITECTURA, INGENIERÍA Y DISEÑO

Máster Universitario en Ingeniería Aeronáutica

Desarrollo de un simulador de vuelo de RPAS multirrotor e integración en un entorno gráfico

Raúl García González

Curso 2022-2023

Título: Desarrollo de un simulador de vuelo de RPAS multirrotor e integración en un entorno gráfico

Autor: Raúl García González

Tutor: Federico Martin de la Escalera

Titulación Máster Universitario en Ingeniería Aeronáutica

Curso 2022-2023

Resumen

Este trabajo propone una implementación completa de un entrenador para aeronaves pilotadas remotamente multirrotor. Para ello, se expone una solución que explota las capacidades de simulación de MATLAB Simulink mediante el desarrollo de una aplicación en Unity que sirve como medio de interacción y visualización con el modelo dinámico y de control, conectando vía UDP la simulación y el visor. Se comienza con una exposición del fondo teórico del modelo dinámico y el control de los aparatos, así como las características y parámetros considerados en la simulación. A continuación, se definen las comunicaciones entre Simulink y Unity y, por último, se describe con detalle el razonamiento, desarrollo y diseño final de la aplicación, describiendo los modelados de drones y entorno pero centrándose en conceptos de experiencia de usuario y diseño de interfaces. En esta implementación, se evalúa asimismo el rendimiento de la solución final para garantizar que la carga computacional de la aplicación sea mínima respecto a la del modelo Simulink.

Palabras clave: Simulación, RPAS, UI/UX, Multirrotor, Matlab Simulink, Unity

Abstract

This work proposes a complete implementation of a trainer for multi-rotor remotely piloted aircraft. To this end, a solution is presented that exploits the simulation capabilities of MATLAB Simulink through the development of an application in Unity that serves as a means of interaction and visualisation with the dynamic model and control, connecting via UDP the simulation and the viewer. It begins with an exposition of the theoretical background of the dynamic model and the control of the devices, as well as the characteristics and parameters considered in the simulation. Next, the communications between Simulink and Unity are defined and, finally, the reasoning, development and final design of the application are described in detail, describing the drone and environment modelling but focusing on user experience and interface design concepts. In this implementation, the performance of the final solution is also evaluated to ensure that the computational load of the application is minimal compared to that of the Simulink model.

Keywords: RPAS, multi-rotor, simulation, UI/UX, Matlab Simulink, Unity

Índice general

Resumen I

Abstract III

1. Introducción 1

1.1. Objetivos 1

1.2. Estado del arte 1

1.2.1. Tecnología 2

1.2.2. Marco Normativo 3

1.2.3. Aplicaciones actuales 3

1.3. Implementación 5

2. Modelo Dinámico 7

2.1. Sistemas de referencia 7

2.2. Modelo de 6GDL 10

2.2.1. Fuerzas y pares internos 10

2.2.2. Fuerzas externas 13

2.3. Modelización de los motores 16

2.3.1. Respuesta del conjunto motor 16

2.3.2. Caracterización de los coeficientes de sustentación 16

2.3.3. Modelización de la batería 16

3. Control 19

3.1. Control en lazo abierto 19

3.1.1. Configuración y geometría del aparato 19

3.1.2. Esquematización de las fuerzas 21

3.2. Control en lazo cerrado 23

3.2.1. Sensores 24

3.2.2. Sistema de control 24

3.2.3. Calibración de ganancias 25

3.3. Funciones adicionales 25

4. Comunicaciones 29

4.1. Entrada de datos: señales de mando 29

4.2. Enlace de datos 30

4.2.1. Enlace de bajada 31

4.2.2. Enlace de subida 31

4.3. Salida de datos: vector de estado y visualización 33

5. Visualización 35

5.1. Motor gráfico 35

5.1.1. Perspectiva (cámaras) 37

5.1.2. Comunicaciones 38

5.1.3. Drones 39

5.1.4. Entorno gráfico 41

5.2. Interfaz de usuario UI/UX 44

5.2.1. Requisitos de la interfaz 45

5.2.2. Diseño de la interfaz 49

6. Resultados 55

6.1. Instalación y requisitos 55

6.2. Limitaciones 56

6.3. Demostración 57

7. Conclusiones **61**

7.1. Trabajo futuro 62

Bibliografía **63**

A. Código **67**

A.1. Código Matlab 67

A.1.1. Constructores 67

A.1.2. Código de lanzamiento de simulación 67

A.1.3. Selección de modo 71

A.1.4. Condición suelo 72

A.2. Código Unity (C #) 73

A.2.1. UpdatePos.cs 73

A.2.2. CommunicationController.cs 74

A.2.3. CameraSwapper.cs 79

A.2.4. Control del viento 81

A.2.5. Giro de las hélices y sonido de motores 85

A.2.6. Interfaz gráfica 87

Lista de figuras

1.1. Mavic 3 2
1.2. Avata 2
1.3. Agras T30 2
1.4. Matrice 300 2
1.5. Ejemplos de la gama de productos de DJI Entreprise 2
1.6. Liftoff: FPV Drone Racing (2018) 4
1.7. AeroSIM RC (2008) 4
1.8. RealFlight RF9.5 Drone Simulator 4

2.1. Comparación del criterio de definición de ejes en el modelo SL vs Unity 10
2.2. Vector de estado del Modelo de 6GDL 11
2.3. Diagrama del flujo de aire sin efecto suelo vs con efecto suelo 14

3.1. Esquema del lazo de control 20
3.2. Configuraciones habituales de multirrotores 20
3.3. Esquematización de las fuerzas y momentos en cada brazo 22
3.4. Flag para aterrizaje en el módulo de motores 26
3.5. Control Manual y Asistido 27
3.6. Control automático: función Hover to Target 27

4.1. Configuración de los ejes del mando 30

4.2. Módulo 1: Recepción de datos 32
4.3. Módulo 1: Recepción de datos 33

5.1. Jerarquía del proyecto de entrenador de RPAS multirrotor Unity 36
5.2. Vista 38
5.3. Localización 38
5.4. Cámara Cinemachine fija a altura de piloto 38
5.5. Vista 38
5.6. Localización 38
5.7. Cámara Cinemachine móvil detrás del dron 38
5.8. Alzado 39
5.9. Perfil 39
5.10. Planta 39
5.11. Vistas del cuadricóptero 39
5.12. Alzado 39
5.13. Perfil 39
5.14. Planta 39
5.15. Vistas del hexacóptero 39
5.16. Alzado 39
5.17. Perfil 39
5.18. Planta 39
5.19. Vistas del octocóptero 39
5.20. Componente BoxCollider para colisiones con el suelo 40
5.21. Luces de navegación en el Hexacóptero 41
5.22. Vista cenital del escenario básico 42
5.23. Modelo 3D de uno de los árboles 43
5.24. Sprites de plantas utilizados 43

5.25. Modelo de manga de viento 43
5.26. Modelo de edificio 43
5.27. Traza de una partícula de viento generada 44
5.28. Las tres capas del diseño de interfaces gráficas 46
5.29. Menú principal 50
5.30. Menú de opciones 50
5.31. Prototipo Inicial 52
5.32. Prototipo Final 52
5.33. Prototipos de interfaz 52
5.34. Diseño Final de la interfaz 52
5.35. Componentes del GameObject UI Canvas 53
5.36. Detalle de elementos de la interfaz 54

6.1. Estructura de la carpeta de la aplicación para su funcionamiento 56
6.2. Requisitos mínimos Unity Player 56
6.3. Requisitos MATLAB R2022b 56
6.4. Cuadricóptero 58
6.5. Hexacóptero 58
6.6. Octocóptero 58
6.7. Consumo de recursos nominal Aplicación vs Modelo MATLAB 59

Lista de tablas

5.1. Ejemplos de posibles intercambios entre el usuario y la aplicación durante el vuelo 47

6.1. Tasas de fotogramas (FPS) de la aplicación (V-Sync activado @144Hz) 58

Capítulo 1

Introducción

1.1. Objetivos

El propósito de este proyecto consiste en desarrollar un entrenador de vuelo para aeronaves tripuladas remotamente (RPAS) multirrotor comerciales, con un entorno gráfico e interactivo que represente de una manera realista las actuaciones y limitaciones de esta clase de aeronaves.

En primer lugar, es importante determinar no solo objetivos realistas, sino también el alcance y profundidad con el que se van a desarrollar. En el caso de este proyecto, los objetivos primarios son los siguientes:

- Estudiar modelos dinámicos preexistentes, desarrollando a partir de ellos un robusto simulador numérico del vuelo de RPAS multirrotor. En concreto, se trabajará sobre los desarrollos anteriores de Linares [1], Timmermans [2], Prol [3] y Fernández [4].
- Desarrollar un entorno de visualización e interacción con el modelo matemático que sea moderadamente realista y permita una experiencia de usuario fiel al vuelo real y útil para el aprendizaje. Esta parte del trabajo se desarrolla desde el principio con el objetivo primario de la usabilidad y la claridad.

1.2. Estado del arte

A fin de realizar un simulador que refleje fielmente la realidad, es en primer lugar necesario hacer un estudio de las soluciones y tecnologías disponibles, así como analizar el marco normativo en el que deben operar estas aeronaves. Por lo tanto, a continuación se expone una breve reseña de la información recabada a este fin:

1.2.1. Tecnología

Con el desarrollo de las tecnologías de RPAS multirrotor, en concreto de capacidades operacionales de productos comerciales, es conveniente adaptar los simuladores y entrenadores para representar las aeronaves reales de manera que mejore tanto la experiencia de usuario (por familiaridad) como la utilidad del mismo (por similitud) en su labor educativa.

Cabe destacar que el mercado de drones se ha beneficiado de las evoluciones tecnológicas de muy diversas áreas, lo que ha permitido un notable crecimiento del mismo en los últimos años [5]. A raíz de esto, el mercado se está especializando para distintos entornos y necesidades operativas, por ejemplo, tomando el ejemplo de la gama ofertada por DJI Enterprise(empresa que supone un 76 % de la cuota de mercado global de drones comerciales y de consumo [5]): se diseñan desde drones comerciales compactos para grabaciones de vídeo como puede ser el “Mavic 3”, drones ágiles para vuelo deportivo FPV (en primera persona) como el “Avata”, hasta grandes drones para la industria agrícola como el “Agras T30” o drones con sofisticados sistemas ópticos para topografía aérea e inspecciones industriales como el “Matrice 300” con LiDAR o sensores térmicos.

Figura 1.1: Mavic 3

Figura 1.2: Avata

Figura 1.3: Agras T30

Figura 1.4: Matrice 300

Figura 1.5: Ejemplos de la gama de productos de DJI Entreprise

Estas aeronaves, mas allá de las diferencias de carga de pago y misiones, también se diferencian en gran medida por su configuración de rotores y sus características de vuelo, lo cual afecta a la manera en la que se pilotan estos aparatos. Además, a nivel de investigación podemos encontrar ejemplos como el dron plegable de la Universidad de Zürich [6] capaz de cambiar su morfología en vuelo para adaptarse a entornos muy reducidos.

En resumen, podría decirse que actualmente nos encontramos ante un mercado en constante evolución, con desarrollos tecnológicos tanto de software como de hardware que deben a su vez adaptarse a las nuevas normativas. Por lo que una herramienta de aprendizaje como la que

se propone en este proyecto deberá ser lo mas modular posible a fin de poder adaptarse a una gran variedad de RPAS multirrotor.

1.2.2. Marco Normativo

Uno de los aspectos a considerar a la hora de cualquier desarrollo en sistemas aéreos es el marco normativo, vital para el correcto funcionamiento de un sistema en el que prima la seguridad. En el caso de los simuladores de drones para entrenamiento, hay que tener en cuenta que aquellas normas que limiten y definan la operación de estos vehículos tendrán que ser reflejadas para una simulación mas realista, por lo que a continuación se hará un resumen del marco normativo actual y sus antecedentes.

Hasta hace apenas unos años, el desarrollo de soluciones comerciales para sistemas aéreos no tripulados, o tripulados remotamente, se había visto limitado, entre otras cosas, por falta de normativa aplicable que delimitara las operaciones para los mismos. Sin embargo, a partir de 2015 con la popularización de multitud de productos en este sector y tras varios incidentes relacionados a la aviación [7], la EASA centró sus esfuerzos en regular los UAS y su integración en el espacio aéreo europeo.

En primer lugar, se publica la A-NPA 2015 [8], estableciendo las directrices regulatorias para la operación de drones, en esta se establecen y definen las tres categorías de operación de drones: abierta, especifica y certificada en función del riesgo asociado. Así mismo, se establecieron grupos de trabajo para estudiar las problemáticas mas incipientes como por ejemplo los incidentes de colisiones de RPAS con aeronaves tradicionales y la limitación geográfica de operación de RPAS.

Estos hechos implicaron la creación de los Reglamentos de la Comisión (EU) 2019/945 [9] y 2019/947 [10], que versan sobre los sistemas aéreos no tripulados y sobre las reglas de operación de sistemas aéreos no tripulados respectivamente. Estas normas entraron en vigor en España el 31 de diciembre de 2020 afectando a todos los drones “independientemente de su uso o tamaño” [11]. Esta es, por tanto, el conjunto de normas mas relevante y actualizado y del cual se extraerán ciertos parámetros a respetar por la simulación.

1.2.3. Aplicaciones actuales

A medida que se ha desarrollado la tecnología asociada a estas aeronaves para aumentar sus capacidades, y con el consecuente desarrollo y limitaciones a su operación, ha crecido la necesidad de entrenamiento para pilotos de RPAS. Y de la misma manera que se han desarrollado

simuladores de vuelo para aeronaves tradicionales, el mercado de los simuladores de RPAS ha visto un importante crecimiento en los últimos años, tanto orientados al ocio desarrolladas ex profeso (Figura 1.6) como orientadas al entrenamiento de pilotos (Figura 1.7 o Figura 1.8).

En el primer caso, podemos ver que se ofrecen unos gráficos vistosos y una simulación física limitada, ya que a fin de crear un juego accesible y divertido se ofrecen varios tipos de "realismo" en cuanto al control del dron se refiere. Así mismo, se muestran entornos realistas, efectos especiales tanto estéticos como de apoyo y cuenta con sistemas de personalización de los distintos drones disponibles que afectan a sus características de vuelo.

Figura 1.6: Liftoff: FPV Drone Racing (2018)

Por otro lado, productos como AeroSIM RC o RealFlight 9.5 ofrecen una estética espartana y una interfaz de usuario (UI) mas completa, dando información sobre el estado de la aeronave y sus sistemas. Los entornos simulados son mas sencillos, con un nivel de realismo aceptable y con el fin de centrarse en programas de entrenamiento y una simulación de vuelo mucho mas cercana a la experiencia real. Cabe destacar que mientras que los simuladores de ocio suelen tener una perspectiva FPV (*First Person View*) desde el dron o una vista en tercera persona de este mismo, los simuladores de entrenamiento mas comúnmente optan por una perspectiva realista en la que la cámara es fija y representa el punto de vista del piloto en tierra.

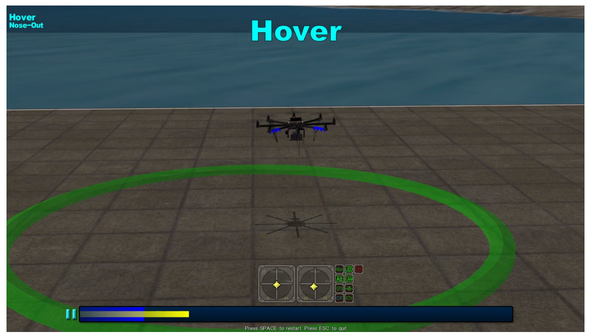


Figura 1.7: AeroSIM RC (2008)

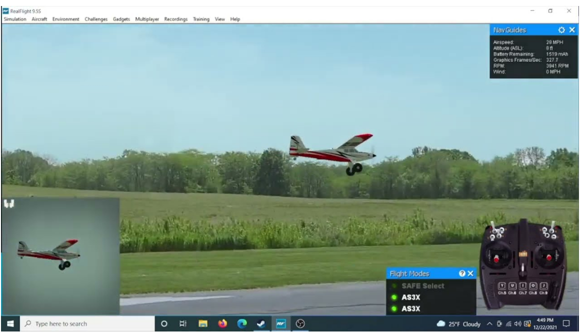


Figura 1.8: RealFlight RF9.5 Drone Simulator

1.3. Implementación

Finalmente y antes de comenzar con el desarrollo, conviene repasar la implementación que se utilizará en este proyecto para alcanzar los objetivos previstos, que consistirá en lo siguiente:

- En primer lugar, y como piedra angular del mismo, se utilizará MATLAB Simulink para realizar en tiempo real todos los cálculos necesarios para determinar el comportamiento del aparato. En este modelo se considerará un modelo dinámico que calcule las fuerzas y momentos aplicables y que pueda de manera determinista calcular la posición y actitud correcta de la nave. Asimismo, esta aplicación deberá gestionar la entrada de datos para las señales de mando y las comunicaciones externas al modelo (telemando y telemetría). Este simulador incluirá por último un sistema de control en lazo cerrado para mejorar la respuesta del mismo ante señales de mando, para este se utilizarán filtros PID con ganancias establecidas para cada modelo de dron.

- En cuanto al sistema de comunicaciones, se selecciona el enlace por UDP. Este protocolo de comunicaciones tiene la ventaja de ser rápido y sencillo y, al hacerse todos los cálculos internamente en MATLAB, la mayor desventaja, que sería la baja garantía de integridad de los datos, no es tan relevante ya que solo afectaría a la visualización de los datos y no a la simulación en si.

- En cuanto a la visualización, así como la interfaz gráfica para el piloto, se ha seleccionado el motor gráfico Unity, que proporcionará una potente colección de herramientas y cuenta con numerosa documentación de referencia. De esta manera, se puede desarrollar un entorno gráfico visualmente vistoso, así como añadir elementos de interfaz gráfica que proporcionen información sobre el estado de la simulación, la aeronave o la entrada de datos. También se busca que esta aplicación sea ligera, para interceder mínimamente en el rendimiento del modelo.

- Por último, como sistema de entrada de datos se utilizará un controlador comercial genérico para PC con el que se pilota. La entrada de datos se realiza e incorpora directamente en MATLAB para reducir al mínimo la latencia de entrada. Así mismo, el entrenador permitirá la interacción de ciertos aspectos del modelo a través de una GUI incorporada en Unity, controlable por teclado y ratón.

Capítulo 2

Modelo Dinámico

En este capítulo se describirá el modelo dinámico que supone el corazón del sistema simulador. En este caso se trata de una parametrización de la dinámica de drones multirrotor integrados en un modelo Simulink, calculando en base a unas fuerzas y momentos (tanto internas como externas al dron) la posición y orientación en el espacio del aparato. En primer lugar, se definirán los sistemas de referencia a usar en el proyecto, así como el modelo de 6 grados de libertad escogido para representar los drones. A continuación, se introducirán las fuerzas y pares internos, generados por el propio dron, y se describirán los distintos efectos externos que se modelan para hacer mas realista la simulación. Por último, se expondrán la modelización de los motores escogida, así como de las baterías para algunos de los drones.

Cabe destacar que el desarrollo de este modelo dinámico está fuera del alcance del presente proyecto salvo pequeñas modificaciones y código de soporte para la simulación, por que lo que a continuación se resume la matemática de este modelo, para mas detalle sobre el desarrollo y código, pueden consultar el trabajo de I. Timmermans [2] y el de A. Fernández [4].

2.1. Sistemas de referencia

En cuanto a los sistemas de referencia que utiliza el modelo, se establecen dos sobre los que se realizarán los cálculos numéricos pertinentes y un tercero necesario para definir correctamente las coordenadas calculadas en el entorno gráfico:

- **Un sistema de referencia de ejes móviles (0):** este se moverá de manera solidaria al dron, con su origen de coordenadas en el centro de masa del dron y los ejes alineados definidos con el criterio definido en esta sección.

- **Un sistema de referencia de ejes fijos (1):** este está fijo al suelo y la posición inicial de cálculo, por tanto solidario a la estación de tierra. Este permitirá ubicar al dron respecto a la posición inicial.
- **Un grupo de sistemas de referencias móviles (u):** en Unity, se define una jerarquía para cada elemento (GameObject) y las coordenadas de cada objeto siempre son relativas al "padre" de este en la jerarquía.

Sistema de referencia de ejes móviles

El criterio escogido para definir estos ejes es el utilizado comúnmente en aviación, donde el eje X tiene valores positivos en la dirección de avance, el eje Y de alinea con el lado derecho de la aeronave y el Z se alinea con la vertical, con valores positivos hacia abajo, conformando el sistema ortogonal definido a derechas.

Asimismo, por comodidad se define el origen de coordenadas de este sistema como el centro de masas del aparato, esto permite simplificar los cálculos significativamente ya que de esta manera la matriz de inercia del sistema móvil será coincidente con la matriz de inercia conocida para los ejes de simetría del dron, la cual puede ser fácilmente obtenida en software CAD (en este caso, SolidEdge) al disponer de un modelo preciso de los drones [12].

Este sistema se utiliza en primer lugar por ser en el que actuará el empuje generado por las hélices del multirrotor y por lo tanto en el que se calcularán las fuerzas y momentos del aparato.

Sistema de referencia de ejes fijos

En cuanto al sistema de referencia de ejes fijos, este se define de manera que sea coincidente en t=0 con el sistema móvil. Éste será la referencia respecto a la cual se medirá la posición y orientación del dron, tal y como se describe en la Figura 2.2. Cabe por lo tanto establecer las matrices de rotación necesarias para cambiar de ejes móviles a fijos, y viceversa, ya que serán necesarios para la correcta parametrización de fuerzas y pares, estas no dejarán de ser mas que la combinación de matrices resultante de giros consecutivos en guiñada (ϕ), cabeceo (θ) y balanceo (ψ) :

$$R(\phi) = \begin{bmatrix} c(\phi) & -s(\phi) & 0 \\ s(\phi) & c(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad R(\theta) = \begin{bmatrix} c(\theta) & 0 & s(\theta) \\ 0 & 1 & 0 \\ -s(\theta) & 0 & c(\theta) \end{bmatrix} \qquad R(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c(\psi) & -s(\psi) \\ 0 & s(\psi) & c(\psi) \end{bmatrix}$$

Por tanto, la matriz de rotación necesaria para pasar de ejes fijos a móviles, así como su inversa,

tras multiplicar las anteriores quedarán como:

$$R = \begin{bmatrix} c(\phi)c(\theta) & c(\theta)s(\phi) & -s(\theta) \\ c(\phi)s(\psi)s(\theta) - c(\psi)s(\phi) & c(\phi)c(\psi) + s(\phi)s(\psi)s(\theta) & c(\theta)s(\psi) \\ s(\phi)s(\psi) + c(\phi)c(\psi)s(\theta) & c(\psi)s(\phi)s(\theta) - c(\phi)s(\psi) & c(\psi)c(\theta) \end{bmatrix} \quad (2.1)$$

$$iR = \begin{bmatrix} c(\phi)c(\theta) & c(\phi)s(\psi)s(\theta) - c(\psi)s(\phi) & s(\phi)s(\psi) + c(\phi)c(\psi)s(\theta) \\ c(\theta)s(\phi) & c(\phi)c(\psi) + s(\phi)s(\psi)s(\theta) & c(\psi)s(\phi)s(\theta) - c(\phi)s(\psi) \\ -s(\theta) & c(\theta)s(\psi) & c(\psi)c(\theta) \end{bmatrix} \quad (2.2)$$

Estas matrices permiten trasladar vectores tanto de posición en XYZ, como velocidades o aceleraciones lineales. Sin embargo, para expresar las velocidades angulares del sistema móvil en el fijo se necesita la matriz Jacobiana invertida (iJ), que se construye considerando la orientación y la posición relativa entre los dos sistemas de referencia. En el caso de un sistema móvil que se mueve con respecto al sistema fijo, la matriz jacobiana dependerá de los ángulos de Euler que describen la rotación entre los sistemas. Esta quedará como:

$$iJ = \frac{1}{cos(\theta)} \begin{bmatrix} 0 & sen(\psi) & cos(\psi) \\ 0 & cos(\psi)cos(\theta) & -sen(\psi)cos(\theta) \\ cos(\theta) & sen(\psi)sen(\theta) & cos(\psi)sen(\theta) \end{bmatrix} \quad (2.3)$$

De esta manera ya se pueden expresar en el sistema fijo o móvil los parámetros asociados al estado del dron en el modelo Simulink.

Sistemas de referencia en Unity

En Unity, debido a su desarrollo como un motor dual para aplicaciones tanto 2D como 3D y a la convención de estos sistemas en distintos programas de animación, define el eje Y en la vertical y, además, es un sistema definido a izquierdas, por lo que el giro en guiñada está invertido respecto al calculado por el modelo. Este criterio se ilustra en la Figura 2.1.

Cabe destacar que, además de lo expresado en la introducción de esta Sección 2.1, en Unity los sistemas de referencia están definidos de manera distinta, por lo que las coordenadas calculadas en Simulink deben ser transformadas para su correcta representación. Esto se hace a la recepción en Unity del vector de estado mediante la siguiente matriz (Ecuación 2.4) para las posiciones mientras que los giros serán idénticos con la salvedad de que $\phi_{SL} = -\phi_{Unity}$.

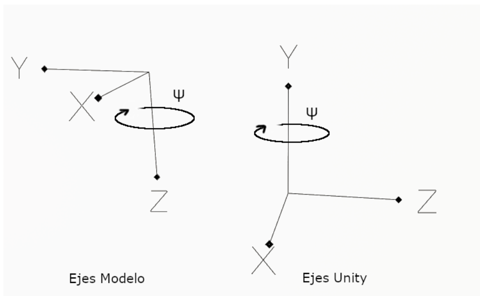


Figura 2.1: Comparación del criterio de definición de ejes en el modelo SL vs Unity

$$R_U = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \tag{2.4}$$

Como se indicó anteriormente, las coordenadas de cada objeto son definidas de manera anidada según la ubicación de cada objeto en la jerarquía, por lo que para facilitar el posicionamiento y rotación de cada dron, se define un GameObject ”Drones”, que se ubicará en la posición inicial del dron y actuará como referencia fija sobre la que se define el movimiento del dron simulado a continuación.

2.2. Modelo de 6GDL

La Figura 2.2 resume, en base a los sistemas de referencia indicados anteriormente, como se define la posición y orientación del dron, determinando de esta manera el vector de estado del modelo dinámico utilizado (Ecuación 2.5). En esta sección de introducen las distintas fuerzas y momentos que permiten determinar con precisión y de manera realista la evolución de estas variables en el tiempo.

$$\bar{x} = \{x,\ y,\ z,\ \psi,\ \theta,\ \phi\} \tag{2.5}$$

2.2.1. Fuerzas y pares internos

Respecto a las fuerzas y pares internos, aquellas que son generados por los rotores y permiten el control del dron mediante la reacción dinámica del dron.

Empuje (T)

Esta es la principal fuerza que permite el vuelo de un dron, y es la resultante de integrar la sustentación generada por las hélices a lo largo de su longitud. Esta fuerza se aplica en la

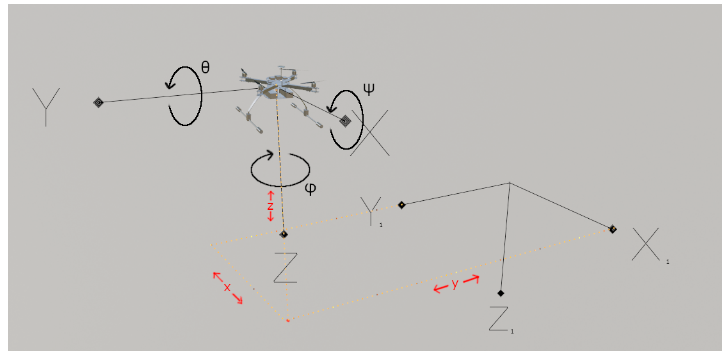


Figura 2.2: Vector de estado del Modelo de 6GDL

dirección vertical local del perfil concreto por definición al ser una de las componentes de la fuerza aerodinámica. Debido a que esta fuerza se genera por la velocidad relativa del perfil respecto del aire, habrá una relación directa entre la velocidad de giro de la hélice (ω) y la fuerza generada(T). Esta puede modelizarse en función de la densidad del aire(ρ), el área barrida por la hélice (A) y su radio(r).

$$T = C_t \, \rho \, A \, r^2 \, \omega^2 \tag{2.6}$$

Donde C_t es un coeficiente de empuje que depende de la geometría de las hélices. Para el propósito de este proyecto, las variables atmosféricas pueden considerarse constantes, ya que es razonable asumir que en el entorno de vuelo de un pequeño multirrotor no se apreciarían diferencias notables, dado que la diferencia de altura respecto al despegue o las velocidades de vuelo no podrían cambiar la densidad del aire significativamente. Al ser la A y r constantes también, la anterior expresión puede por lo tanto resumirse como:

$$T = c_t \, \omega^2 \tag{2.7}$$

Esta nueva c_t se puede obtener experimentalmente, según [12], para el conjunto motor-hélice de cada dron como se expone en la Sección 2.3.

Momento de arrastre (Q)

De la misma manera que se genera una fuerza vertical, la sustentación, el movimiento relativo de la hélice respecto al aire genera una fuerza de resistencia, y al integrar esta para toda la hélice se genera un momento de arrastre, Q, sobre el rotor en dirección contraria al giro de la hélice. De manera análoga al empuje, se puede modelizar este momento en función de la

velocidad angular de la hélice.

$$Q = -C_q \, \rho \, A \, r^3 \, \omega^2 \tag{2.8}$$

La suma de aporte de estos momentos por cada rotor supondrá un momento τ_z en el eje vertical del aparato. Es por esto, como se explicará mas en profundidad en el Capítulo 3, que los rotores se distribuyen de manera simétrica y con sentidos de giro opuestos en cada par de rotores para cancelar, en el caso estacionario, este momento. Así mismo, las variaciones de velocidad angular de cada rotor respecto de su simétrico es lo que permitirá tener mando sobre el eje guiñada.

El coeficiente C_q, de manera análoga al C_t es una constante que depende de la hélice, y al considerarse constantes las demás variables, se simplifica la ecuación, pudiendo obtener experimentalmente esta c_t como se expone en la Sección 2.3.

$$Q = -c_q \, \omega^2 \tag{2.9}$$

Par de alabeo y cabeceo (τ_{xy})

Respecto al mando sobre los ejes de inclinación del dron, el cabeceo y el balanceo, este se obtiene mediante el momento que genera el empuje sobre el centro de masas del dron al estar separado del mismo. En este caso, se calcula para cada rotor como el producto vectorial de la distancia al centro de masas (d_i) por la fuerza T descrita anteriormente y por al tener la fuerza T solo componente en z, podrá descomponerse la misma en los ejes de balanceo τ_x y cabeceo τ_y.

$$\tau_{xy} = d_i \times T_i \tag{2.10}$$

En este punto es donde acabarían los efectos que compondrían el comportamiento ideal del modelo, es decir, aquellos que solamente están influenciados directamente por el efecto del empuje generado por cada uno de los motores. Este comportamiento, al tener una modelización mas sencilla, es el que se utilizará para determinar el control de la aeronave, como se expondrá en el Capítulo 3. Sobre estos, se añaden efectos secundarios internos y externos que permiten elevar el realismo de la simulación, acercándose así a una representación realista de la dinámica del dron.

Efecto giroscópico

Al girar a altas velocidades, el efecto giroscópico se manifestará cuando haya un cambio en la velocidad de giro de las hélices o cuando se aplique un par de giro externo al dron. Este efecto es de baja influencia en comparación a los momentos anteriormente mencionados, pero debido a la conservación del momento angular, se generará en cada uno de los brazos del dron un momento giroscópico que tenderá a resistir dicho cambio o par de giro. Este par se producirá en el eje perpendicular tanto al giro relativo del aparato como al eje de rotación de la hélice. Por lo tanto, para cada rotor quedará como:

$$\bar{\tau_i}' = -I_r\left(\bar{\omega}_i \times \bar{\Omega}\right) \tag{2.11}$$

Donde I_r es el momento de inercia de la hélice, Ω es la velocidad angular del dron y ω_i es la velocidad de giro de la hélice en cuestión. El valor de I_r puede obtenerse teóricamente sabiendo la geometría de la hélice en el caso de una pala real, pero al disponer del modelo CAD de las hélices, estas se pueden obtener directamente del modelo a través de SolidEdge para este proyecto. Sin embargo, se aprecia que, al considerar la distribución antisimétrica de sentidos de giro de los rotores, a priori se cancelarán entre sí los efectos producidos por cada par de rotores. Este efecto por lo tanto, solo será significativo cuando simultáneamente haya una diferencia de giro en mas de un rotor y se produzca un giro relativo del aparato, pudiendo afectar solo moderadamente a la estabilidad en maniobras agresivas. Además, debido a que la rotación de las hélices ($\bar{\omega_i}$) está limitada al eje vertical, la resultante suma de todos los rotores tendrá componentes en x e y de la siguiente forma:

$$\tau_x' = -I_r\Omega_y \sum \omega_i \tag{2.12}$$

$$\tau_y' = -I_r\Omega_x \sum \omega_i \tag{2.13}$$

2.2.2. Fuerzas externas

Además de las fuerzas internas del dron, para simular correctamente su comportamiento habrá que modelar los efectos externos mas notables, como por ejemplo el efecto suelo, el rozamiento con el aire o la influencia del viento en los diversos cálculos llevados a cabo.

Efecto Suelo

En primer lugar, este es el efecto que experimenta el aparato al volar a poca distancia de la superficie y será especialmente notable por tanto en despegues y aterrizajes. En concreto, se

trata de un aumento de la sustentación consecuencia de la interacción entre el flujo de aire generado por los rotores del dron y la superficie cercana. Cabe destacar, que aunque supone una ayuda en despegues, al requerir menos energía para generar la misma sustentación y elevar el vuelo, el efecto suelo genera flujos de aire turbulento bajo el dron, lo que puede afectar negativamente a la respuesta del dron dificultando aterrizajes precisos, por ejemplo.

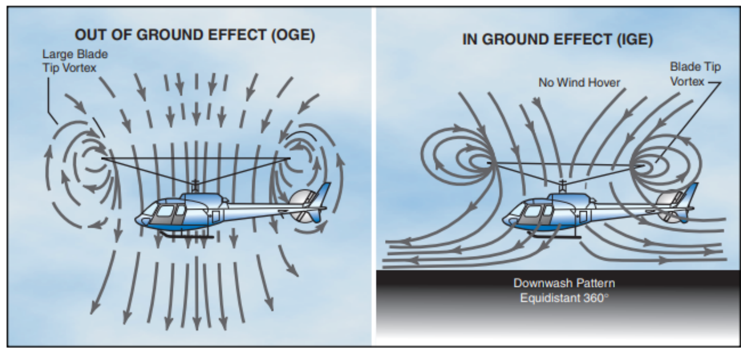


Figura 2.3: Diagrama del flujo de aire sin efecto suelo vs con efecto suelo [13]

Este efecto está muy estudiado en el caso de las aeronaves de ala rotatoria tradicionales como son los helicópteros, y su traslado a multirrotores solo recientemente ha sido estudiado, por ejemplo, por Sanchez-Cuevas [14], que establece una modelización mas precisa que la dada por la teoría de helicópteros y además, estudia los casos de efecto suelo parcial, cuando solo uno de los rotores está por encima de una superficie. Esto, sin embargo, excede el alcance del proyecto, por lo que se usará siguiente relación para el efecto suelo:

$$\frac{T_{suelo}}{T_{libre}} = \frac{1}{1 - \frac{r^2}{16z^2}} \tag{2.14}$$

Lo anterior, a pesar de ser una simplificación, se considera suficiente para generar un resultado realista, ya que en nuestro caso para afinar el modelo sería necesario contar con datos empíricos para cada una de las plataformas simuladas en función del número de rotores. En cuanto a los límites en los que actúa esta fuerza, se establece en [14] que por debajo de un radio este modelo no es válido ya que la perturbación generada por el flujo turbulento cerca del suelo invalida el modelo de empuje desarrollado en el Apartado 2.2.1, y, en el extremo opuesto, este aumento de empuje deja de ser significativo a partir de 5r.

Rozamiento con el aire y modelización del viento

En cuanto efectos externos, quizás uno de los mas significativos, especialmente dadas las características de los drones a simular, sea el viento. El modelo matemático cuenta con la posibili-

dad de introducir viento (en forma de un vector de velocidades) externamente. Para el presente proyecto, esta información se genera en Unity y se envía a través de UDP. Una vez recibida, se utiliza para añadir una fuerza externa debida a la fricción del aire con el dron. Esta fuerza será proporcional al cuadrado de velocidad relativa entre el aire y el dron, por lo que se sumarán las velocidades calculadas y del viento para obtener este dato.

$$v_{rel} = v_{viento} - v_{dron} \tag{2.15}$$

$$F_{viento} = C_d \, \rho \, A \, v_{rel}^2 \tag{2.16}$$

Sin embargo, la estimación de manera empírica de este coeficiente de rozamiento no es trivial ya que incluso de tener acceso a los medios experimentales necesarios, este dependerá de muchos factores difícilmente controlables durante el vuelo real pudiendo llevar a dificultad a la hora de evaluar los resultados [15]. Una manera de estimar este parámetro es mediante simulación CFD utilizando los modelos geométricos, mientras que una alternativa a esto sería mediante comparación con aeronaves similares.

Condiciones ambientales

Con respecto a las condiciones ambientales, como pueden ser las atmosféricas como presión, temperatura o densidad del aire, pero también la fuerza de la gravedad local, afectan a la dinámica del dron y a las fuerzas generadas como se vio anteriormente. En el modelo se considera que el entorno de operación del dron es limitado en tiempo y espacio, por lo que estás variables se consideran constantes a lo largo de la simulación, este supuesto. Sin embargo, se mantiene la posibilidad de modificar la gravedad, densidad del aire y viscosidad del aire en el constructor, pudiendo, de ser necesario, emular otro entorno de operación (por ejemplo alta montaña) si fuera necesario. Dentro del alcance de este proyecto se mantendrán constantes con los siguientes valores:

```
%% Entorno
    %Gravedad
        dron.g   = 9.81;
    %Densidad del aire
        dron.rho = 1.184;
    %Viscosidad del aire
        dron.muv = 1.5e-5;
```

2.3. Modelización de los motores

Una parte crucial para caracterizar correctamente los aparatos simulados es proporcionar una representación realista del funcionamiento de los motores, por lo que se utilizará una combinación de desarrollos teóricos, datos experimentales y de diseño de los motores utilizados en los drones reales a simular para generar un modelo convincente de los mismos. Tendrá por tanto que modelizarse su respuesta, sus características aerodinámicas y su consumo de batería. Cabe destacar que este desarrollo no es el propósito de este proyecto, por lo que se expone aquí tan solo de manera descriptiva, pudiendo encontrar mas detalles sobre la teoría y ensayos llevados a cabo en el trabajo de A. Fernández [4].

2.3.1. Respuesta del conjunto motor

Los motores tipo *brushless* controlados electrónicamente utilizados en estos aparatos no son capaces de una respuesta inmediata y variaciones de velocidad angular instantáneas, por lo que es necesario modelizar la respuesta transitoria ante un cambio de velocidad requerida. Cabe destacar que este tipo de motores vienen controlados por un variador de velocidad que modificará la señal dada a los motores, sin embargo, esta modelización queda fuera del alcance e interés de este proyecto. Por lo tanto, se puede considerar el conjunto motor-controlador como un sistema de primer orden con un tiempo característico (t_i) de 0.1 segundos. De esta manera, se define por lo tanto para cada motor un sistema como el siguiente:

$$G(s) = \frac{1}{1 + t_i s} \tag{2.17}$$

Esta información se proporciona al modelo por parte de los constructores (Ver [4]).

2.3.2. Caracterización de los coeficientes de sustentación

Estos pueden obtenerse de manera teórica siguiendo desarrollos derivados del estudio de helicópteros como propone Timmermans [2], sin embargo, dada la naturaleza de nuestro proyecto, en el que se tiene acceso al *hardware* real, es posible obtener estos valores experimentalmente mediante la elaboración de curvas de T vs ω con el conjunto motor-hélice real fijo sobre una bancada. De esta manera obtiene los datos correspondientes al hexacóptero y octocóptero Alejandro Fernández [4], que serán los utilizados en este proyecto.

2.3.3. Modelización de la batería

En cuanto al consumo de batería, utilizando la misma bancada anterior puede medirse la corriente requerida por el motor para alcanzar RPM, generando así una curva de I vs ω que, una

vez tabulada, permitirá mediante interpolación medir el consumo instantáneo de cada uno de los motores.

Esta señal puede entonces ser integrada en el tiempo para obtener la energía consumida (en amperios hora). Así mismo, se utilizan los datos de los respectivos fabricantes respecto a la capacidad de las baterías presentes en cada aparato, de esta manera es sencillo obtener el estado de carga de las baterías, esta información será la que se proporcionará como información al piloto a través de UDP.

Capítulo 3

Control

Mientras que en el capítulo anterior se ha establecido el comportamiento dinámico del dron ante las fuerzas y momentos aplicados sobre el mismo, en este se detallan los principios que se han seguido para establecer el lazo de control, así como funciones de control adicionales que se han implementado para vuelo asistido y automático, de manera que se amplíen las posibilidades de pilotaje y se facilite la interacción del piloto con el aparato. El controlador actuará de interfaz entre las órdenes de vuelo (la entrada de referencia) y los motores (actuadores), siendo a su vez realimentado por la orientación medida por la IMU (sensores), cerrando así el lazo de control realimentado clásico. Asimismo, esta parte del modelo Simulink permite establecer como se comportarán los drones multirrotor con número de rotores mayor que cuatro al perder uno de ellos (fallo de rotor). Una vez más, como en el capítulo anterior, la descripción de la teoría y sistemas implementados se hará de manera descriptiva, para mas detalles, consultar el trabajo de Alejandro Fernández [4].

Pese a que cabe destacar que dada la naturaleza modular de este entrenador, este control deberá ser particularizado y calibrado para cada uno de los drones a simular, este capítulo se centrará en describir la filosofía de diseño y matemática del sistema en función de el número de rotores N cada uno con una distancia $\bar{d}_i$ al centro de masas.

3.1. Control en lazo abierto

3.1.1. Configuración y geometría del aparato

Un factor importante a la hora de diseñar, y por tanto de modelizar y simular, drones multirrotor es la configuración de los distintos rotores y la geometría resultante. En la actualidad, existen

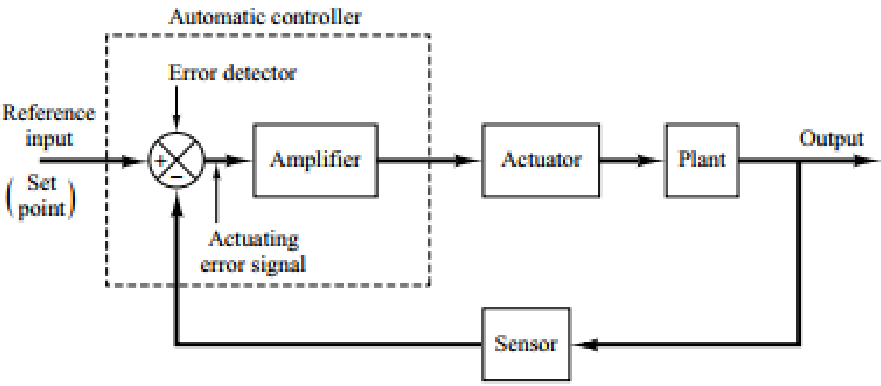


Figura 3.1: Esquema del lazo de control

[16]

multitud de configuraciones posibles, siendo lo mas habitual las soluciones con configuración de rotores estáticas y coplanarias. Por ejemplo, para los cuadricópteros existen dos configuraciones típicas: en “+” y en “x” con rotores fijos [17], pero también se pueden desarrollar cuadricópteros controlables en configuración “Y” similares a los tricópteros pero instalando 2 rotores con sentidos de giro opuestos en la cola [18] o con cuatro rotores coplanarios en “+” o en “x” pero con capacidad de bascular, de manera que el dron pueda volar paralelo al suelo en todo momento [19]. De manera similar, para hexacópteros y octocópteros existen tanto configuraciones coplaniaras como diversas combinaciones de rotores coaxiales o rotores inclinables. La elección

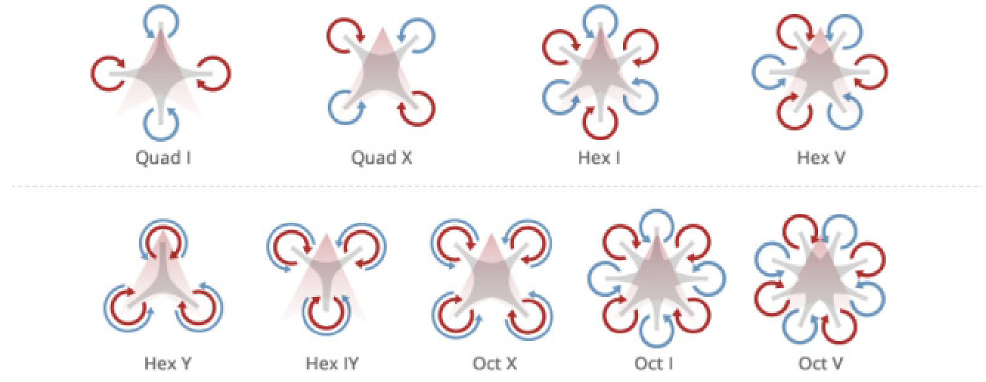


Figura 3.2: Configuraciones habituales de multirrotores

[17]

de la configuración en este caso viene determinada por el diseño de los drones disponibles, al estar intentando simular drones reales, por lo que las configuraciones escogidas se resumen en

la siguiente lista:

- **Cuadricóptero**: Configuración en “X”, con el primer brazo a 45º respecto al eje x y el resto distribuidos uniformemente.
- **Hexacóptero**: Configuración en “V”, con el primer brazo a 30º respecto al eje x y el resto distribuidos uniformemente.
- **Octocóptero**: Configuración en “I”, con el primer brazo sobre el eje x y el resto distribuidos uniformemente.

A partir de esto, podemos empezar por definir la matriz de distancias D, que tendrá dimensión $3 \times N$, en la que se almacenará la posición de cada motor por columnas, expresadas todas en el sistema de referencia móvil definido anteriormente.

$$D = \begin{bmatrix} d_1 cos(\alpha_1) & d_2 cos(\alpha_2) & \dots & d_N cos(\alpha_N) \\ d_1 sen(\alpha_1) & d_2 sen(\alpha_2) & \dots & d_N sen(\alpha_N) \\ h_1 & h_2 & \dots & h_N \end{bmatrix} \quad (3.1)$$

Donde d_i es la magnitud de la distancia horizontal del centro de masas al rotor, α_i es el ángulo del brazo respecto al eje x y h_i la altura del rotor respecto el centro de masas. Debido a la configuración definida anteriormente, se puede determinar que h_i y d_i serán constantes e iguales para cada rotor, mientras que α_i puede expresarse en función de N de manera genérica como:

$$\alpha_i = \alpha_0 + \frac{i-1}{N} 2\pi \quad (3.2)$$

Con esto, la matriz de distancias quedará como:

$$D = \begin{bmatrix} dcos(\alpha_0) & dcos(\alpha_0 + \frac{1}{N}) & \dots & dcos(\alpha_0 + \frac{i-1}{N}) \\ dsen(\alpha_0) & dsen(\alpha_0 + \frac{1}{N}) & \dots & dsen(\alpha_0 + \frac{i-1}{N}) \\ h & h & \dots & h \end{bmatrix} \quad (3.3)$$

3.1.2. Esquematización de las fuerzas

Con el fin de desarrollar la matriz de control, se valorarán, como se menciono en el capítulo anterior, tan solo las fuerzas y momentos principales: el empuje generado por el motor T_i y la contribución en momentos de esta fuerza en τ_x, τ_y y $Q_i = \tau_z$, haciendo uso en este caso de la matriz D anterior para descomponer la $\tau_x y$ descrita previamente. Quedando por tanto las cuatro fuerzas y momentos en función de la velocidad angular de la hélice ω_i como se puede apreciar

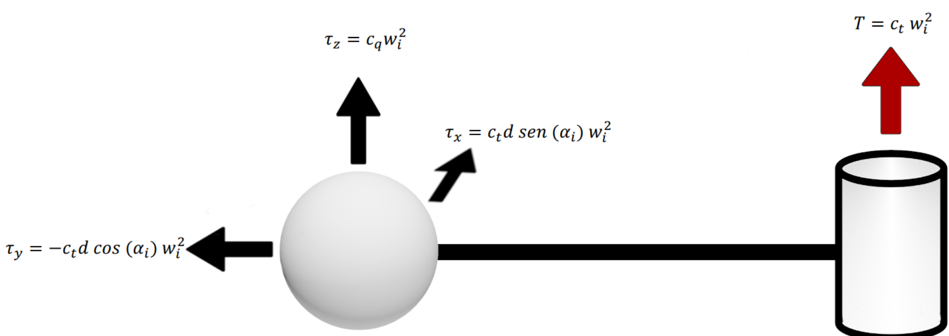


Figura 3.3: Esquematización de las fuerzas y momentos en cada brazo

en el esquema de la Figura 3.3. En base a este esquema, se puede entonces particularizar para cada brazo y generar la suma de fuerzas y momentos que actúan sobre el dron:

$$\begin{pmatrix} T \\ \tau_x \\ \tau_y \\ \tau_z \end{pmatrix} = \begin{pmatrix} c_t & c_t & c_t & \dots & c_t \\ c_t dsen(\alpha_1) & c_t dsen(\alpha_3) & c_t dsen(\alpha_3) & \dots & c_t dsen(\alpha_N) \\ -c_t dcos(\alpha_1) & -c_t dcos(\alpha_2) & -c_t dcos(\alpha_3) & \dots & -c_t dcos(\alpha_N) \\ -c_d & c_d & -c_d & \dots & -c_d\, sgn(\omega_N) \end{pmatrix} \begin{pmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \\ \dots \\ \omega_N^2 \end{pmatrix} \tag{3.4}$$

Esta ecuación en esencia será la que representa el control de la aeronave, mientras que las señales de $(T, \tau_x, \tau_y, \tau_z) = H$ suponen la señal de mando, aquella que queremos obtener mediante la entrada del piloto, la matriz de control A supone la conversión de estas señales a las velocidades angulares necesarias en cada motor. De esta manera, el control de la aeronave se obtendrá realizando la pseudoinversa de la matriz $4 \times N$ de control y resolviendo para $\bar{\omega}$ en la siguiente ecuación:

$$H = A\bar{\omega}^2 \tag{3.5}$$

Sobre esta matriz de control cabe destacar que a pesar de ser un modelo de 6GDL, solo se cuenta con cuatro entradas (las del vector H), lo que por naturaleza supondrá un sistema infra-actuado. Sin embargo, como se demuestra en [12], controlar un sistema de 6GDL será posible siempre que el sistema sea diferencialmente plano, esto es, que se puedan expresar todas las posibilidades del vector de estado y la entrada en función de la "solución plana" y sus derivadas, esto es, expresado en función de cada combinación de (x, y, z, ϕ) y sus derivadas en nuestro

caso. Al comprobar este hecho se demuestra que a partir de 4 actuadores (es decir, el cuadrirrotor) un multirrotor tiene suficiente mando para controlar el sistema. Sin embargo, hay que indicar que esta solución no sería válida en caso de que estuviéramos a 90º de balanceo o cabeceo, ya que perderíamos la determinación del eje de guiñada, entrando en lo que se conoce como *gimbal-lock*, por suerte, dada la naturaleza del proyecto, no se espera este tipo de maniobras en los drones por lo que no es necesario adaptar la matemática.

En resumen, para el caso del lazo directo de control, la solución del sistema para el control de los motores por lo tanto es la siguiente:

$$\bar{\omega}^2 = A^{-1}\,H \tag{3.6}$$

De la misma manera que se cuenta con un sistema infraactuado pero controlable en el caso del cuadricóptero, para el hexacóptero y el octocóptero tendremos la ventaja de disponer de mas grados de libertad de los necesarios, lo que permite no solo disponer de mayor potencia (al tener mas motores) si no de mayor flexibilidad en el control y mejor resiliencia a fallos ya que incluso en casos de fallo motor el sistema seguiría siendo controlable. Pudiendo de esta forma aterrizar de manera segura incluso tras fallo motor. Estos fallos motores están incluidos en el modelo sustituyendo la columna correspondiente de la matriz A por ceros en caso de fallo para ese motor.

3.2. Control en lazo cerrado

Dada la compleja y no lineal relación entre las entradas del piloto (vector H) y el vector de estado de salida ($\bar{x}$), a pesar de garantizar la controlabilidad, el pilotaje puede no ser intuitivo con tan solo el control en lazo abierto, por lo que es conveniente realimentar la señal de referencia con el estado actual, de manera que se simule la posible realimentación mediante sensores con las que contaría un dron moderno. Por lo tanto, en primer lugar se expondrá el tipo de sensores con los que contaría una plataforma de estas características, así como lo que se ha modelizado en este caso. Posteriormente, y una vez establecido el sistema retroalimentado mediante las señales de estos sensores simulados, corresponde introducir brevemente el tipo de control utilizado y como las distintas señales se ajustan mediante ganancias.

3.2.1. Sensores

En una plataforma tipo multirrotor, lo mas típico es contar con sensores tipo MEMS que compongan una sencilla IMU de pequeño consumo y buenas prestaciones para medir la orientación del aparato. Estas IMU basadas en sensores MEMS generalmente componen un conjunto de giróscopos, acelerómetros y, opcionalmente, magnetómetros [20]. De esta manera, son capaces de determinar la vertical local (midiendo aceleración debida a la fuerza de la gravedad con los acelerómetros) e incluso determinar la guiñada midiendo el campo magnético terrestre mediante el uso de los magnetómetros. Sin embargo, debido al ruido de alta frecuencia presente, integrar estas señales para obtener la posición no proporciona valores realistas [21], por lo que generalmente se acompañan estos con sensores GNSS capaces de determinar posición y velocidad por satélite, ya sea para corregir la señal de la IMU mediante fusión de sensores o como sistema independiente.

En el presente proyecto, se modeliza tan solo una sencilla IMU capaz de realimentar posiciones y velocidad angulares (para realimentar los ejes de balanceo, cabeceo y guiñada), así como velocidades lineales (para el avance en x o y, así como el ascenso/descenso). A este fin, se asumen unos “sensores perfectos” que proporcionan el valor actual del vector de estado medido en ejes móviles sin contar con ruido, latencia o imprecisión alguna.

3.2.2. Sistema de control

En cuanto al control de estas señales para la correcta respuesta del sistema, se hace uso de distintos sistemas de control para cada eje de mando. El objetivo de esto es que cada una de las señales de mando de la respuesta esperada con en un tiempo mínimo y minimizando posibles oscilaciones, alcanzando el valor estacionario de manera óptima. A continuación, se introduce estos sistemas de control:

- **Control de empuje:** Este se retroalimenta la velocidad vertical y se calibra para quedar en equilibrio ante una posición neutral del eje de empuje en el mando. En cuanto al control se opta por un control proporcional que permite un amortiguamiento a la señal.
- **Control de guiñada:** La guiñada se controla mediante realimentación de la velocidad de giro y un control proporcional.
- **Control de balanceo y cabeceo:** dado que el propósito de las inclinaciones respecto al plano horizontal es el de desplazarse, la señal del mando de balanceo y cabeceo se retroalimenta con las velocidades lineales correspondientes y en cuanto al control se aplica

control PD.

3.2.3. Calibración de ganancias

Cabe destacar por lo tanto que con los sistemas de control descritos anteriormente, será necesario determinar en total 13 ganancias para cada modelo de dron, este trabajo se hace manualmente para ajustar el control a lo que sería esperado. De esta manera, se llega a un control afinado de las distintas variables de control del sistema. Estas se establecen en el momento de lanzar la simulación (Código en Anexo A, Apartado A.1.2).

3.3. Funciones adicionales

Además del vuelo manual controlado, expuesto anteriormente, se han añadido ciertas funciones de vuelo asistido y automático, que sin llegar a ser un piloto automático completo, permiten operar el dron de manera alternativa. Estas opciones pueden ser seleccionadas en tiempo real desde el visor y se envía la selección vía UDP al modelo Simulink en la forma de una sola variable que permite elegir el modo de operación, esta solución permitiría en el futuro añadir tantas funciones como fueran necesarias. Esta variable a su vez será convertida en distintas flags para activar y desactivar los códigos correspondientes en el modelo, como puede verse en Anexo A, Apartado A.1.3. La lista de funciones implementadas, además del control manual retroalimentado descrito anteriormente, es la siguiente:

- **Vuelo asistido nivelado seleccionando direcciones cardinales de avance:** en este modo, se selecciona vuelo hacia delante/atrás o derecha/izquierda y se modifican las entradas de los canales de balanceo y cabeceo correspondientemente para moverse en esa dirección. En este modo, la entrada de referencia para los ejes de cabeceo, balanceo y guiñada quedarán fijos a la orden establecida mientras que el eje de empuje queda libre, por lo que se puede añadir potencia en caso de ser necesario para subir o bajar. En esta situación por ejemplo, si seleccionamos el avance lateral a la izquierda, se establecerá una referencia de cabeceo y guiñada de 0 mientras que el balanceo tendrá un valor fijo que producirá una traslación a la izquierda en el aparato según lo establecido anteriormente.
- **Hover to target:** modo automático en el que se define una altura y el aparato subirá o bajará lo necesario para ir a esa altura y quedará fijo en la misma. En este modo no se tiene en cuenta las entradas del piloto.
- **Autoaterrizaje:** en este modo asistido se limitan las RPM de los motores para asegurar

que el aparato baje a una velocidad segura y controlada. Además, una vez alcanza el suelo se apagan los motores. Esta función supone el *Flight Termination System* (FTS) del modelo, un componente importante de cara a la operación de drones en la actualidad. Además, esta función está configurada de manera que se active automáticamente bajo ciertas circunstancias que requieran la finalización del vuelo, como batería baja ($SoC < 15\,\%$), entrada en zonas restringidas o perdida de señal de mando.

Para implementar estos modos de operación en el modelo, en primer lugar tenemos un selector de modo entre automático y manual/asistido, generando así dos lazos distintos diferenciados por la entrada de referencia (Figuras 3.5 y 3.6). Cabe destacar que el modo de autoaterrizaje se selecciona por separado y se aplica en el controlador del lazo directo (Figura 3.4) y no entra en la retroalimentación, ya que fijará las RPM a un valor preestablecido para cada uno de los drones.

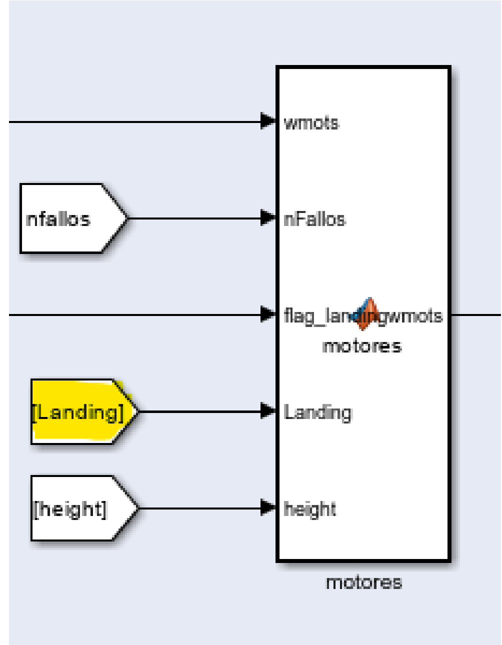


Figura 3.4: Flag para aterrizaje en el módulo de motores

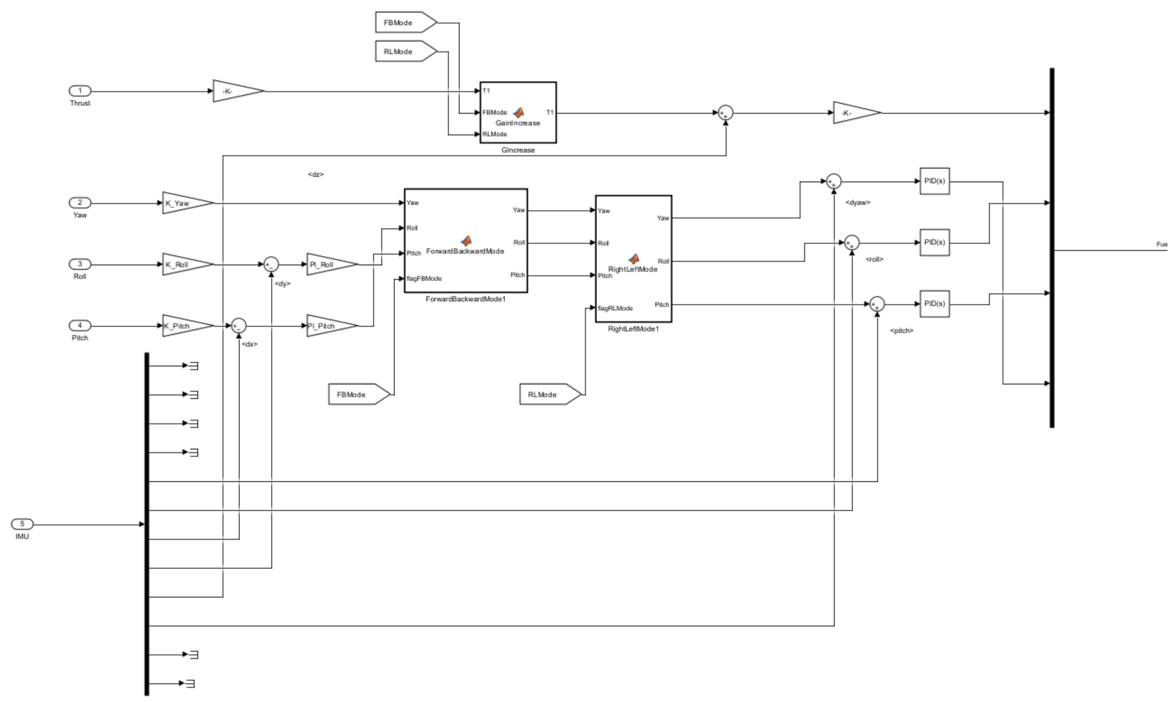


Figura 3.5: Control Manual y Asistido

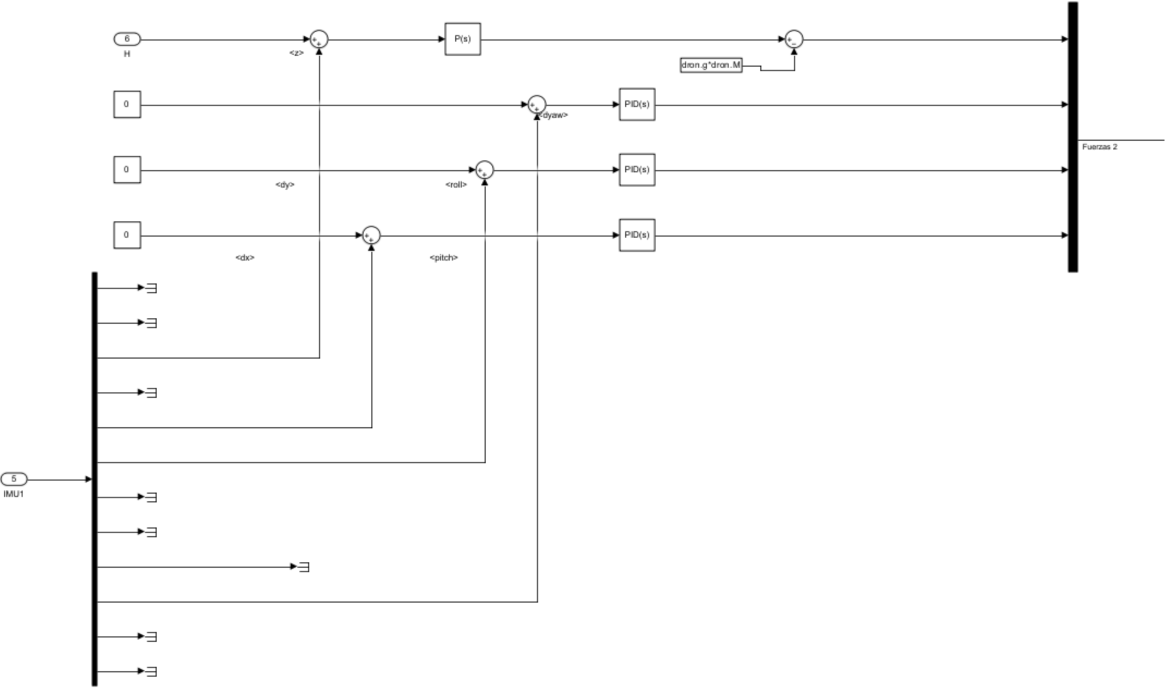


Figura 3.6: Control automático: función Hover to Target

Capítulo 4

Comunicaciones

Otro aspecto que debe tenerse en cuenta al respecto de la simulación, al ser un proyecto que utiliza distintas plataformas para el cálculo matemático del modelo y la visualización, que tiene el propósito de entrenamiento en tiempo real, es que las distintas partes del sistema: usuario, modelo matemático y entorno gráfico, han de estar conectados de manera efectiva, comunicándose de manera precisa y rápida.

Es por ello que en este capítulo se exponen los diferentes enlaces de datos, así como la información transmitida por el usuario y entre Simulink (el modelo matemático) y Unity (el entorno gráfico).

4.1. Entrada de datos: señales de mando

La primera etapa de comunicación es la que existe entre el usuario y el modelo matemático, que supone la señal de mando que controla el dron. Esta se consigue mediante la entrada de un mando XInput directamente en el Simulink para reducir al mínimo la latencia de entrada. El uso de un mando genérico permite cierta flexibilidad a los posibles usuarios y, además de las señales de mando, permite tener acceso a varios botones que se utilicen para distintas opciones dentro de la simulación. Para hacer uso de este controlador, se utiliza la librería de Mathias Voigt [22]. Así mismo, se definen definirán los ejes de control como se puede ver en la Figura 4.1.

Por otra parte, cabe destacar que esta es la “entrada primaria” que se hace directamente en Simulink y permite el control directo del dron, sin embargo, ciertos controles del entorno de simulación se podrán configurar desde el visor, por lo que existirá otro tipo de “entradas secundarias”, estos se estudian en profundidad en la Sección 5.2 y consisten en una serie de menús

de opciones y botones interactuables.

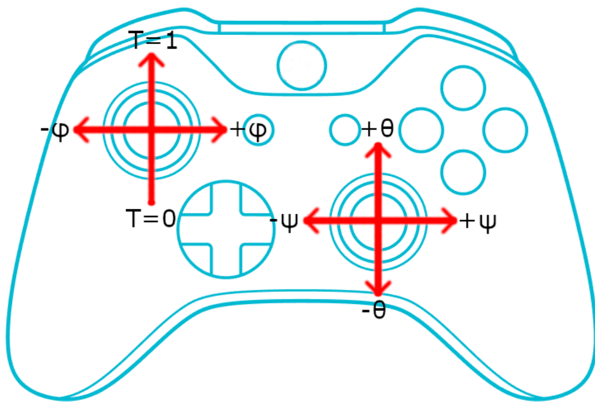


Figura 4.1: Configuración de los ejes del mando

4.2. Enlace de datos

Para comunicar los dos programas principales (el modelo de Simulink y el programa en Unity) se utiliza UDP (*User Datagram Protocol*). Este es un protocolo mínimo de nivel de transporte orientado a mensajes que permite la comunicación rápida y sencilla alternativa al protocolo TCP [23].

A diferencia de este último, el protocolo UDP no requiere establecer una conexión previa y al no contar con complejos mecanismos de control de errores ni de comprobación de transmisión permite mandar datos a mayor velocidad y con menor latencia. Para el presente uso, esto es ideal, ya que las desventajas de este protocolo, como son la perdida de paquetes y que no se tenga confirmación de recepción de los mismos no es crítica, ya que no afecta a la simulación en si, tan solo a la visualización y, sin embargo, sí que es crítico el disponer de un sistema que responda rápidamente y tenga un retraso mínimo entre las entradas de mando y la representación visual de la respuesta.

Respecto al apartado visual, definido en el siguiente capítulo, se ha desarrollado un sistema agnóstico al modelo, de manera que pueda operar en base a distintos modelos matemáticos siempre que reciba los mismos datos básicos del vector de estado de la aeronave y sea capaz de realimentarle ciertos parámetros y medidas. Además, esto tiene la ventaja de ser un código aplicable de igual manera a los distintos drones incluidos. En ese sentido este enlace de datos estaría formado por, esencialmente, dos vectores, el correspondiente al enlace de bajada (del modelo al visor) y el correspondiente al enlace de subida (del visor al modelo).

4.2.1. Enlace de bajada

Este enlace supone el medio por el cual el modelo comunicará al entorno gráfico lo necesario para representar la simulación. En concreto, se envía un vector de datos de doble precisión (double en MATLAB) con los siguientes datos:

- **Tiempo:** se proporciona el valor del tiempo de simulación.
- **Posición:** se dan las coordenadas X Y Z del dron calculadas por el modelo en ejes fijos.
- **Actitud:** se proporcionan los valores de cabeceo, guiñada y balanceo del dron calculados por el modelo, esto es el giro de los ejes cuerpo respecto los ejes fijos.
- **Velocidad en ejes móviles:** se utiliza como medida de la velocidad de vuelo del dron para dar mas información al piloto a través de la interfaz.
- **Estado de Carga (SoC) de la batería:** utilizando la modelización de consumo en función de las RPM, se calcula el valor de la carga restante en el modelo y se manda a Unity para su representación.
- **Rad/s:** se envían los datos de velocidad angular de cada motor, ajustando en función del número de rotores del dron en cuestión. Se envían en rad/s por comodidad, ya que así se calculan en el modelo y así se define la velocidad angular de las palas en Unity.

4.2.2. Enlace de subida

Este enlace tiene por fin proporcionar al modelo con datos sobre factores externos al dron (afín al enlace que sería la telemetría en un modelo HITL) y configuraciones/parámetros seleccionados en la simulación. Por ejemplo, las condiciones respecto al terreno, al ser este un elemento no simulado por el modelo Simulink o el dron y condiciones de viento elegidas. Asimismo, cabe destacar que, como se puede ver en el Figura 4.3, en este módulo será donde ubiquemos el bloque "Real-Time Sync" que permite a un modelo Simulink ejecutarse en tiempo real independientemente de la velocidad de cálculo del equipo, en cuanto al tiempo de muestreo, se define como $dt = 0.0025s$.

- **Fuerza y dirección del viento:** esta es seleccionable desde la aplicación, por lo que hay que realimentarla al modelo Simulink.

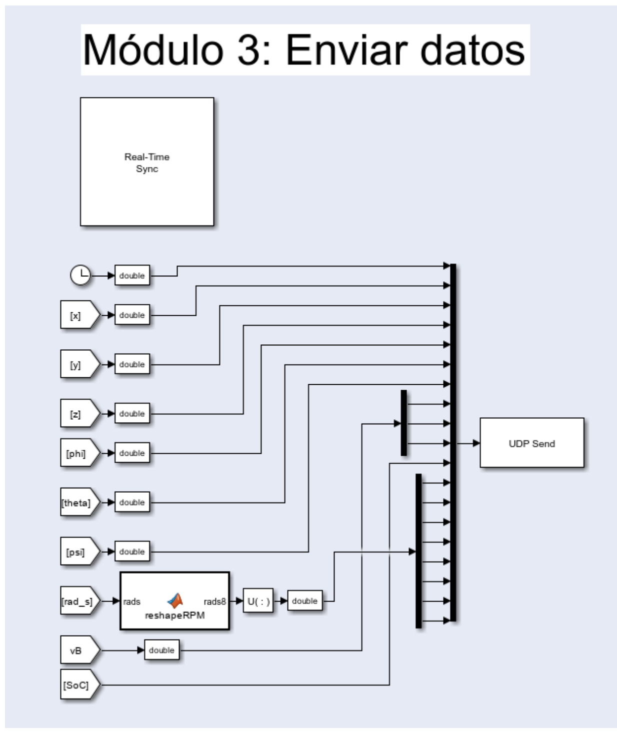


Figura 4.2: Módulo 1: Recepción de datos

- **Altura sobre el suelo (AGL):** al ocurrir la simulación en un entorno sintético, es importante determinar la altura sobre el terreno, tanto por información para el piloto como para realimentar el modelo y calcular por ejemplo el efecto suelo.
- **Orden de parada y arranque de la simulación:** se diseña la aplicación en Unity de manera que pueda lanzar e interrumpir la simulación
- **Fallos de rotores:** variables nfallos y RotorFallido, que determinarán el número de motores perdidos y por cual empezó.
- **Funciones adicionales:** Se definen varios modos de operación Manual, Hover, Flight Asssit, con la posibilidad futura de implementar más.
- **Autoaterrizaje:** como medida de Flight Termination System (FTS), se establece una orden que indica al dron descender hasta llegar al suelo.

Llama la atención que no se retroalimente la selección de dron, ya que es una de las variables fundamentales de este entrenador, sin embargo, esta no se enviará vía UDP, utilizándose una herramienta de Unity para escribir en los registros de Windows que se explicará en el Capítulo 5.

Una limitación significativa de la implementación con un modelo Simulink es que no es posible ejecutar la simulación directamente desde el visor, lo que supone la necesidad de disponer de un ordenador con licencia MatLab/Simulink para poder hacer uso de esta herramienta. Como línea de trabajo futuro, sería interesante compilar este modelo de Simulink en forma de código (ya sea C/C++ como soporta el Simulink Embedded Coder u otro) para poder ejecutar el modelo matemático en el visor, siendo necesario por tanto solo el ejecutable creado en Unity para hacer uso del simulador.

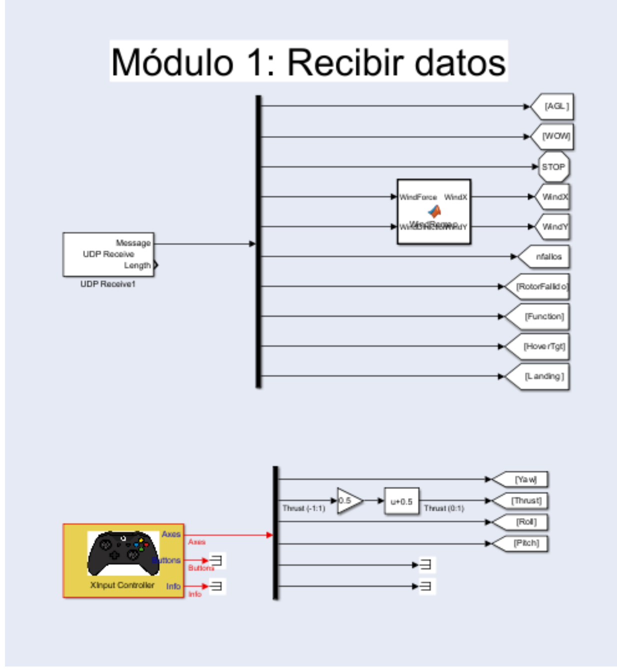


Figura 4.3: Módulo 1: Recepción de datos

4.3. Salida de datos: vector de estado y visualización

Por último, la información que genera el modelo debe comunicarse visualmente al piloto para poder interactuar con el modelo de manera similar a la realidad, esto se desarrollará en detalle en el siguiente capítulo y consistirá en la utilización de los vectores de estado y telemetría anteriores para representar en tiempo real el estado de la simulación en un entorno gráfico mediante una interfaz de usuario.

Capítulo 5

Visualización

Este capítulo se centrará en el desarrollo del visualizador y entrenador basado en el modelo matemático descrito anteriormente. Para ello, se utilizará el Motor Unity, una popular herramienta elegida por su flexibilidad, accesibilidad y capacidad para crear entornos suficientemente realistas de manera relativamente sencilla.

En primer lugar, se exponen las ventajas de Unity para la simulación de drones y como se utilizan sus herramientas especificas para las necesidades del proyecto. Después, se describen los distintos elementos incorporados como parte de la simulación y por último se explica en detalle los requisitos, elementos y diseño de la interfaz gráfica y experiencia de usuario (UI/UX).

5.1. Motor gráfico

El Editor de Unity será la herramienta utilizada en este proyecto para la elaboración del entorno gráfico y la interfaz de usuario del modelo. Esta permite la integración y representación de la información generada por el modelo matemático descrito anteriormente, actuando como medio de comunicación visual con el piloto y ofreciendo una manera de modificar condiciones de la simulación.

En primer lugar, y fruto de su uso primario como herramienta de desarrollo de videojuegos, Unity proporciona un entorno de desarrollo intuitivo y accesible, con una amplia gama de herramientas y recursos que facilitan la creación de contenido interactivo. Además, proporcionará un buen rendimiento gracias a su optimización y permitirá integrar fácilmente diversas funcionalidades que mejoren el realismo del modelo mediante una retroalimentación a Simulink, mientras que su potente motor de renderizado permiten representar de manera precisa el vuelo y la interac-

ción de los drones. Por último, cabe destacar que sus capacidades multiplataforma permitirían adaptar el entrenador a una amplia gama de dispositivos, aunque en este caso estaría limitado por la necesidad de disponer de Matlab Simulink.

De cara a exponer los distintos elementos, cabe introducir el sistema de jerarquía de GameObjects en Unity. Este sistema es fundamental para organizar y estructurar los elementos de una escena. En Unity, un GameObject es el nombre que recibe de cualquier objeto en la Escena, y la jerarquía permite establecer relaciones padre-hijo entre ellos. La jerarquía se representa como un árbol, donde un GameObject puede tener uno o varios hijos, y cada hijo puede tener sus propios hijos. Esto crea una estructura jerárquica anidada que facilita la manipulación y la interacción entre los GameObjects. Por ejemplo, si se tiene un objeto que representa un dron y se quiere añadirle hélices, se pueden crear los GameObjects necesarios como hijos del dron principal y colocar las hélices en ellos. A este fin, la jerarquía juega un papel importante, pues la transformación de los objetos, tanto en posición, como rotación y escala, es heredada de padres a hijos. Es decir, si mueves el dron las hélices que generamos como hijos de este se moverán de manera solidaria. Por tanto, las posiciones, rotaciones y escalas de cada GameObject estarán siempre expresadas como relativas a su padre (o a la conjunción de padres en caso de GameObjects anidados).

Además, la jerarquía facilita la organización y la gestión de los componentes y scripts asociados a los GameObjects. Se puede, por ejemplo, asociar un script a un GameObject y este puede acceder a variables y componentes localizados en GameObjects hijos. Esto permite crear comportamientos complejos mediante la combinación de scripts en diferentes niveles de la jerarquía.

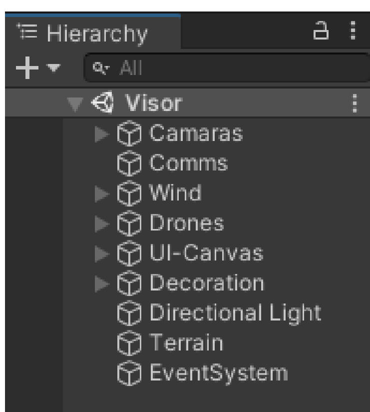


Figura 5.1: Jerarquía del proyecto de entrenador de RPAS multirrotor Unity

En esta sección se aprovechará esta jerarquía (Figura 5.1) para ir destacando algunos de los elementos incorporados en el proyecto y su implementación concreta.

5.1.1. Perspectiva (cámaras)

Un punto a determinar es la perspectiva ("*Point of View*" o POV) desde la que queremos ver el aparato y el entorno que estamos simulando. En este apartado se dispone de varias alternativas, por ejemplo:

- **Cámara fija que siga al dron:** esta es quizás la mas relevante al contexto de aplicación, permite ver al dron en todo momento en pantalla desde la perspectiva que tendría un piloto de pie. Esta cámara tiene las mismas limitaciones que se tendrían en la vida real en cuanto a la dificultad de operar BVLOS ("*Beyond Visual Line of Sight*"). Para el objeto de este proyecto, la cámara será estática al estar simulando un entorno de operaciones relativamente reducido.
- **Cámara en primera persona (FPV):** este tipo de cámaras son las utilizadas para drones deportivos que se operan BVLOS. En este tipo de perspectiva, el piloto obtiene el contexto visual de la ubicación y dirección del dron a través de una cámara instalada en el propio dron
- **Cámara en tercera persona de seguimiento del dron:** una cámara que siga de cerca y desde detrás al dron en todo momento es quizás la opción menos realista ya que no es viable su implementación en drones reales. Sin embargo se considera útil para tener un control mas delicado de la aeronave y observar de cerca sus actuaciones en un contexto educativo.

En nuestro caso se ha optado por dos de estas: la cámara fija (simulando un piloto de pie) y la cámara en tercera persona por su valor didáctico al poder ver de cerca las actuaciones de la aeronave. La cámara FPV, quizás mas orientada a drones deportivos y de competición, queda por tanto fuera del alcance previsto en este proyecto. Además, de los drones simulados, tan solo uno dispone de gimbal sobre el que montar dicha camara como veremos a continuación.

En cuanto a la implementación, se utiliza el paquete Cinemachine, que es una colección de herramientas para Unity que se utiliza para crear y controlar cámaras virtuales en aplicaciones como la nuestra. La utilidad principal de Cinemachine radica en su capacidad para gestionar de forma automática y dinámica la composición de la cámara, el seguimiento del objetivo, los movimientos de cámara suaves, las transiciones y más. Por lo tanto, tan solo hay que indicar la configuración de estas cámaras, a que elemento deben seguir, con que distancia y el compositor de la cámara virtual automáticamente se trasladará y rotará en base a esto a medida que se

mueve el elemento en cuestión. En ese sentido, se genera el sencillo código “CameraSwapper” que en primer lugar determina el dron que hemos escogido simular (para hacerlo el “objetivo” de la cámara) y posteriormente se encarga de configurar correctamente las distancias y perspectivas de seguimiento en base a esta elección. Asimismo, desde este script se lanza el cambio entre las dos cámaras, leyendo el estado de un botón a tal efecto el script solo desactiva un GameObject y activa el otro, siendo el compositor de Cinemachine responsable de transicionar suavemente entre los dos puntos de vista. En las siguientes figuras (5.4 y 5.7) se muestran las perspectivas escogidas y el posicionamiento de las cámaras relativo a la posición inicial del dron, mientras que en el Anexo A, Apartado A.2.3 podrá consultarse el script correspondiente.

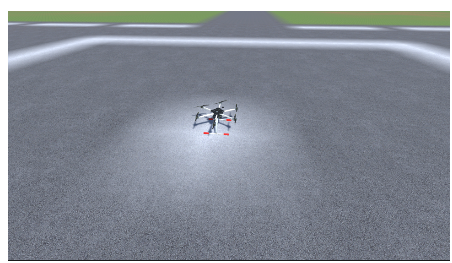


Figura 5.2: Vista

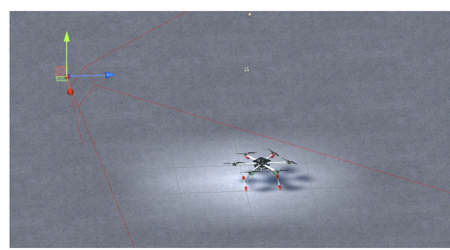


Figura 5.3: Localización

Figura 5.4: Cámara Cinemachine fija a altura de piloto

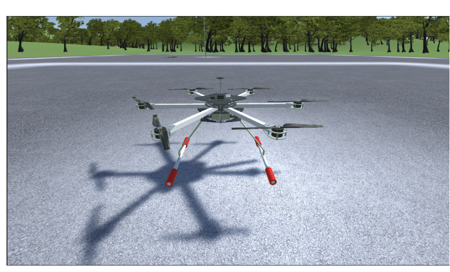


Figura 5.5: Vista

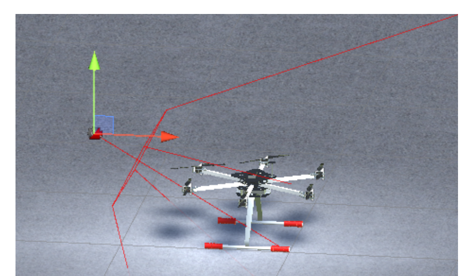


Figura 5.6: Localización

Figura 5.7: Cámara Cinemachine móvil detrás del dron

5.1.2. Comunicaciones

En el capitulo anterior(4) se exponen las comunicaciones realizadas desde Simulink, con los enlaces de bajada y subida correspondientes. En el lado Unity, estas comunicaciones se concentran a través del objeto “Comms”, que incluye el script “Communication Controller”, así como los scripts UDPReceiver y UDPTransmitter, basados en el trabajo de Cihad Dogan [24]. Este script genera los vectores de subida y bajada descritos anteriormente descritos y los almacena en variables públicas para el acceso de los demás componentes del sistema, puede consultarse

en el Anexo A, en el apartado A.2.2.

5.1.3. Drones

El entrenador cuenta con cuatro modelos de RPAS multirrotor, seleccionables desde el menú principal. Los modelos han sido proporcionados por Federico Martin de la Escalera, exportados desde SolidEdge y configurados en Unity. Se les han aplicado texturas para hacerlos mas realistas y, por ejemplo, diferenciar la parte anterior y posterior de cada aparato.

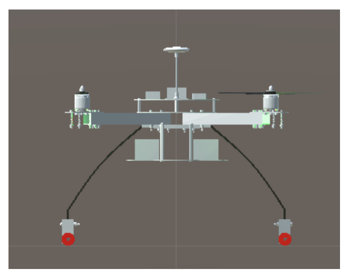


Figura 5.8: Alzado

Figura 5.9: Perfil

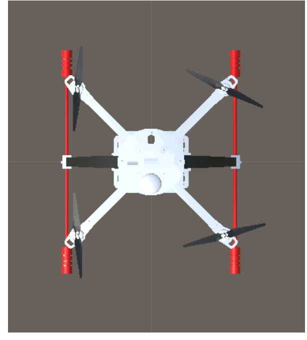


Figura 5.10: Planta

Figura 5.11: Vistas del cuadricóptero

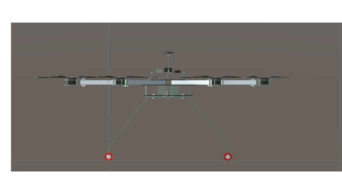


Figura 5.12: Alzado

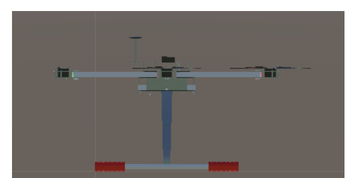


Figura 5.13: Perfil

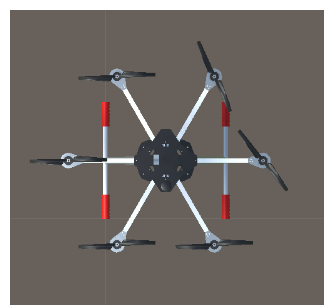


Figura 5.14: Planta

Figura 5.15: Vistas del hexacóptero

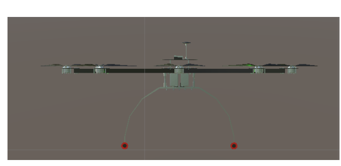


Figura 5.16: Alzado

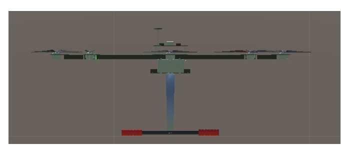


Figura 5.17: Perfil

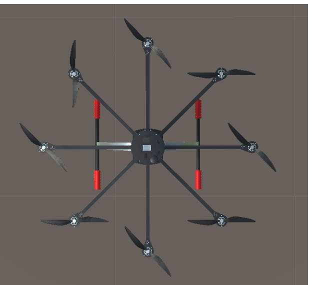


Figura 5.18: Planta

Figura 5.19: Vistas del octocóptero

Efectos

Sobre estos modelos de los drones se aplicarán diversos efectos visuales y sonoros que representen su interacción con el mundo y den feedback visual al piloto a fin de hacerlo mas realista. En concreto, se le aplican los siguientes:

- **Giro de las hélices y sonido de los motores:** utilizando la información de las RPM recibida por UDP, se hace girar las hélices a la velocidad angular correspondiente, además se modula el tono del sonido utilizado de manera que se perciba sonoramente las subidas y bajadas de RPM de los motores. Puede verse el código para estas dos funciones anteriores en el Anexo A, Apartado A.2.5. El componente utilizado en Unity para reproducir el sonido permite asimismo establecer como varía el volumen con la distancia a la cámara, por lo que permitirá también, en el caso de la perspectiva del piloto fijo, contar con cierto feedback sonoro sobre la distancia a la que vuela el aparato.

- **Colisiones con el suelo y el entorno:** utilizando el componente "BoxCollider" de Unity, podemos detectar cuando el dron entra en contacto con cualquier otro sólido rígido (definido por su componente "RigidBody"), esto se utilizará para determinar la condición de peso en ruedas que posteriormente se manda a través de UDP al modelo Simulink. Se denota aquí que este se define con cierto margen a fin de permitir un mayor rango de actuación en la simulación, ya que de ser mas fino si el dron volase rápido hacia el suelo podría resultar que atravesara completamente el suelo de un paso de simulación al siguiente, produciéndose una colisión no detectada. El resultado final de como el modelo detecta la colisión y como se reacciona en ese caso se desarrolla en mayor profundidad en el Anexo A, Apartado A.1.4.

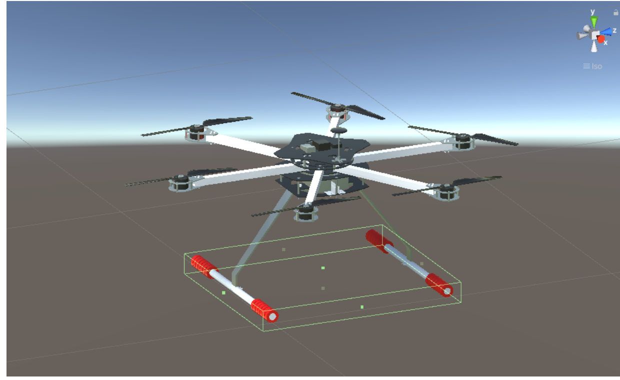


Figura 5.20: Componente BoxCollider para colisiones con el suelo

- **Luces de posición:** se incorporan cuatro puntos de luz en cada dos, dos verdes en los brazos frontales y dos rojas en los brazos traseros, esto es afín a las luces LED presentes en muchos drones y ayudan a determinar orientación del aparato desde tierra además de mejorar la visibilidad del mismo. Para implementarlos, bastó con añadir objetos con el componente “Point Light” en el borde de los brazos correspondientes para cada uno de los drones.

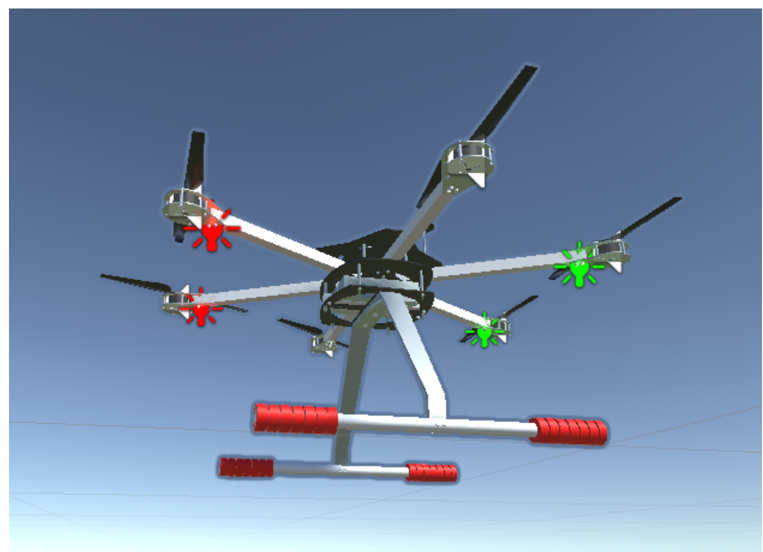


Figura 5.21: Luces de navegación en el Hexacóptero

5.1.4. Entorno gráfico

Dado el alcance y propósito del proyecto, el entorno sintético no requiere una representación de un lugar real, por lo que se opta por un terreno y decorados sencillos que sirvan como referencia pero no distraigan del objeto: aprender a volar los drones. En concreto, se ha desarrollado un escenario básico que sería adecuado para entrenamiento inicial, pudiendo diseñarse escenarios mas complejos para entrenamientos mas avanzados. A continuación se describen brevemente los elementos incorporados y ciertos efectos incluidos.

Terreno

En cuanto al terreno, se utilizan las herramientas de Unity para definir una pequeña zona de operaciones en una zona arbolada. Asimismo, se define una zona de operaciones asfaltada y plana donde, a priori, se llevará a cabo el vuelo.

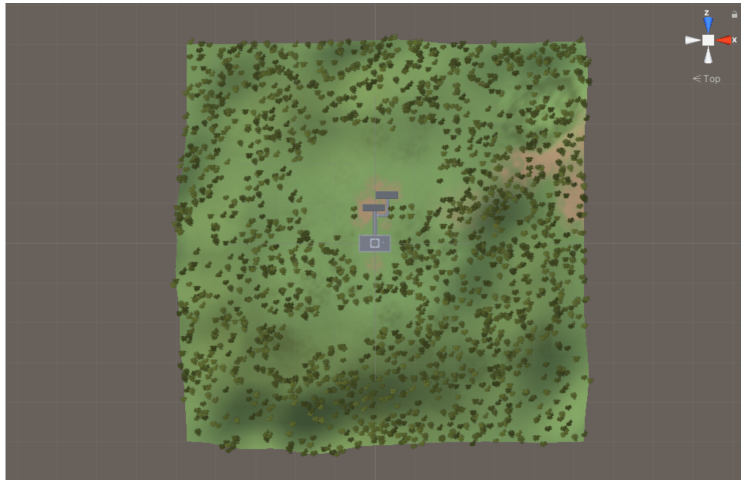


Figura 5.22: Vista cenital del escenario básico

Decorados

Asimismo, para poblar este terreno se incluyen una serie de elementos de decorado (Figuras 5.23 a 5.26) que si bien no afectan significativamente al vuelo, proporcionan un entorno natural pseudo-realista que permita poner en contexto al aparato. Además, se han habilitado las físicas de Unity para que algunos de los elementos se vean afectados por el viento, dando mayor feedback al piloto de la presencia de viento. Entre ellos se encuentran los siguientes:

- **Árboles y follaje:** se utilizan paquetes de assets proporcionados por la comunidad de Unity de manera gratuita ([25] y [26]) para poblar las inmediaciones del campo de vuelo. Estos se integran a través de la herramienta dedicada para ello en Unity, que permite "pintar" sobre el terreno una distribución aleatoria de árboles con variedad en las alturas. Además, para reducir el impacto al rendimiento, estos utilizan un efecto de "billboarding" que hará que a partir de cierta distancia, los árboles se rendericen como una imagen 2D en lugar del modelo 3D completo. De manera similar se añaden distintas texturas para añadir detalles al suelo, flores, plantas bajas y hierbas.

- **Manga de viento:** se ha creado una manga de viento para ofrecer una representación diegética de la dirección y velocidad del viento, utilizando Blender para la malla 3D y el componente "Cloth" de Unity para simular su movimiento. Esta permite dividir la malla en ciertas secciones deformables, restringiendo la distancia máxima a la que cada nodo de la malla puede moverse, lo que permite generar zonas mas rígidas que otras.

- **Edificios:** de cara a aumentar ligeramente la complejidad del escenario, se incluyen varios edificios sencillos, creados una vez mas en Blender. En Unity estos se comportan como

obstáculos sobre los que el dron pueda volar o aterrizar al haberse añadido los componentes “Collider” correspondientes.

Figura 5.23: Modelo 3D de uno de los árboles

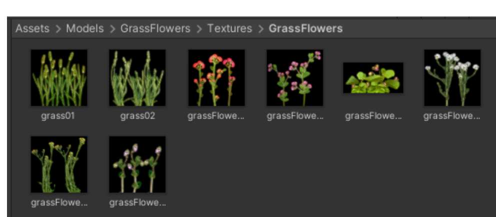


Figura 5.24: Sprites de plantas utilizados

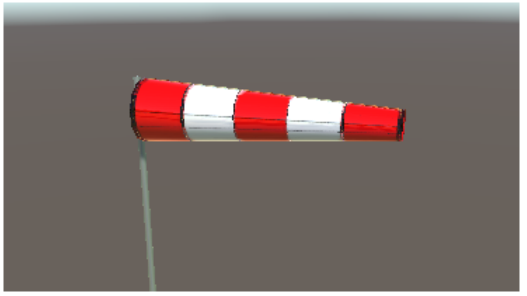


Figura 5.25: Modelo de manga de viento

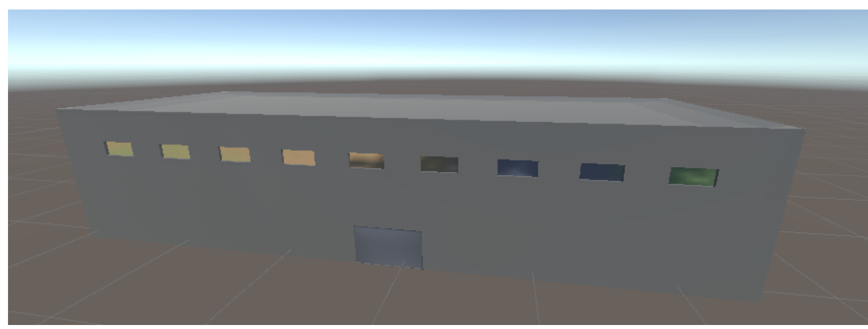


Figura 5.26: Modelo de edificio

Efecto del viento

A continuación se expone la representación del viento en el entorno y su efecto sobre los elementos anteriormente mencionados. Para ello, en primer lugar se ha de tener en cuenta como está modelizado el viento en el modelo Simulink, en este caso, un campo de velocidades uniforme en el espacio con componentes en X e Y. Convenientemente, para representar este efecto

Unity cuenta con los componentes tipo “WindZone”, que permiten crear zonas de viento con diferentes características, como dirección, velocidad y turbulencia, que afecten a los objetos y elementos del escenario. Al añadir este componente a un objeto en el editor de Unity, se puede establecer cómo el viento interactúa con los demás elementos, y está específicamente diseñado para afectar a árboles y detalles de un GameObject de terreno y sistemas de partículas, ambas funcionalidades serán aprovechadas en este proyecto.

La zona de viento se definirá en base a dos parámetros en el menú de opciones de la simulación, una dirección y una velocidad de viento, como se verá en el apartado siguiente sobre la interfaz. Una vez definida esta velocidad del viento, debido a que las zonas de viento de Unity se definen en base a la fuerza, se estima la fuerza generada por el mismo en base a la presión dinámica A.2.4, de manera similar a como se realiza en el modelo Simulink.

Para que la manga de viento reaccione a la velocidad y dirección del viento se dispone del script “WindSockRotate”(Anexo A, Apartado A.2.4) que establece la rotación y fuerza con la que ondea esta manga en función del viento.

Aparte de la manga de viento, se incluye un sistema de partículas que, aunque no sea realista, permite una visualización del vector del viento activo en todo momento, incluso sin la manga de viento visible. Este sistema de partículas seguirá al dron y emitirá elementos visuales que flotarán en la dirección actual del viento, dejando una traza visible tal y como se aprecia en la Figura 5.27 y de esta manera sirve de indicación adicional para el usuario.

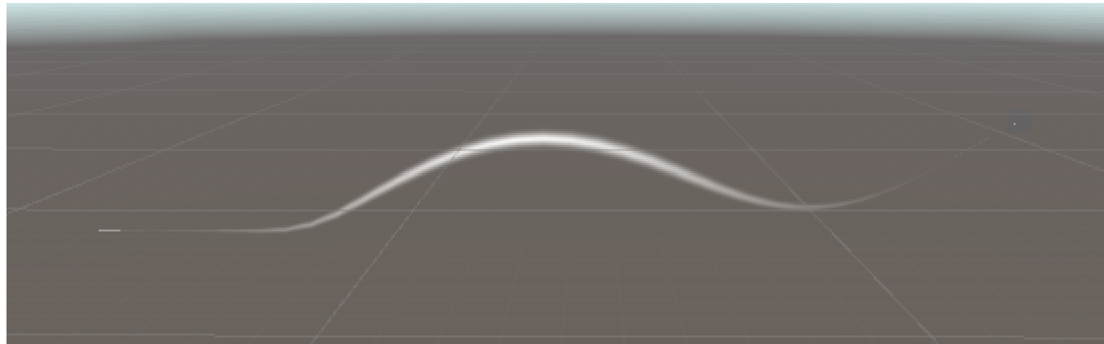


Figura 5.27: Traza de una partícula de viento generada

5.2. Interfaz de usuario UI/UX

El propósito de la interfaz gráfica en este entrenador es doble: en primer lugar, se busca establecer el vínculo computador-usuario de manera efectiva y eficiente, transmitiendo de manera

clara y concisa la información necesaria; por otra parte, se busca cierto nivel de inmersión que permita al usuario tener una experiencia lo mas cercana posible al vuelo real. Estos dos principios a priori pueden ser contradictorios, por lo que será necesario determinar una solución de compromiso entre el realismo y la comunicación efectiva de la información.

Entrelazado en esta dicotomía entre realismo y efectividad, se plantea el desafío de generar un entrenador intuitivo, que permita a un novato descubrir y entender el funcionamiento de un dron sin demasiada experiencia previa. A este fin, el diseño de la interfaz ha de ser necesariamente sencilla, que no simple, reduciendo en la medida de lo posible su intromisión en la experiencia. Por lo tanto, nos centraremos tanto en diseñar la interacción del usuario con la simulación, como en diseñar la experiencia de usuario de manera que se guíe visualmente al usuario y pueda aprender a utilizar la herramienta sin necesidad de explicación previa. Esto es lo que Norman [27] expresa como “capacidad de descubrimiento” de un diseño, las limitaciones deberían por tanto ser evidentes y debería proporcionarse el feedback adecuado al usuario de manera que pueda aprender intuitivamente y fomentando el aprendizaje kinestésico al utilizar la herramienta.

Por ejemplo, uno de los puntos importantes a la hora de aprender el manejo de multirrotores es que, mientras que la dinámica de un coche puede ser intuitiva, las ecuaciones que gobiernan el movimiento de un dron y la perspectiva que tiene el usuario (fijo en tierra, con el aparato capaz de ejecutar maniobras complejas en el aire) dificultan el “entendimiento natural” de como se comportará la aeronave, por lo tanto, las funcionalidades de la interfaz gráfica deben estar al servicio del usuario, aliviando esta brecha de conocimiento.

Asimismo, no solo es importante la funcionalidad y la manera de interactuar de la interfaz, si no que ha de tenerse en cuenta la distribución de los elementos y su diseño para garantizar una experiencia de usuario efectiva. Para ello, se siguen principios de diseño de interfaces de aplicaciones del libro de O’Reilly [28], teniendo en cuenta que la audiencia objetivo es alguien con interés en el manejo de drones, pero quizás no gran interés ni trasfondo físico-matemático. De esta manera, esta sección de la memoria se orientará en base a las tres capas de diseño (ver Figura 5.28): que información se quiere intercambiar con el usuario, como se ejecuta esta transacción y por último, como se presenta esta información al usuario.

5.2.1. Requisitos de la interfaz

Información requerida

La información requerida se puede agrupar en varias categorías: la información sobre el estado de la simulación, la información sobre el vuelo, la información sobre el entorno en el vuela el

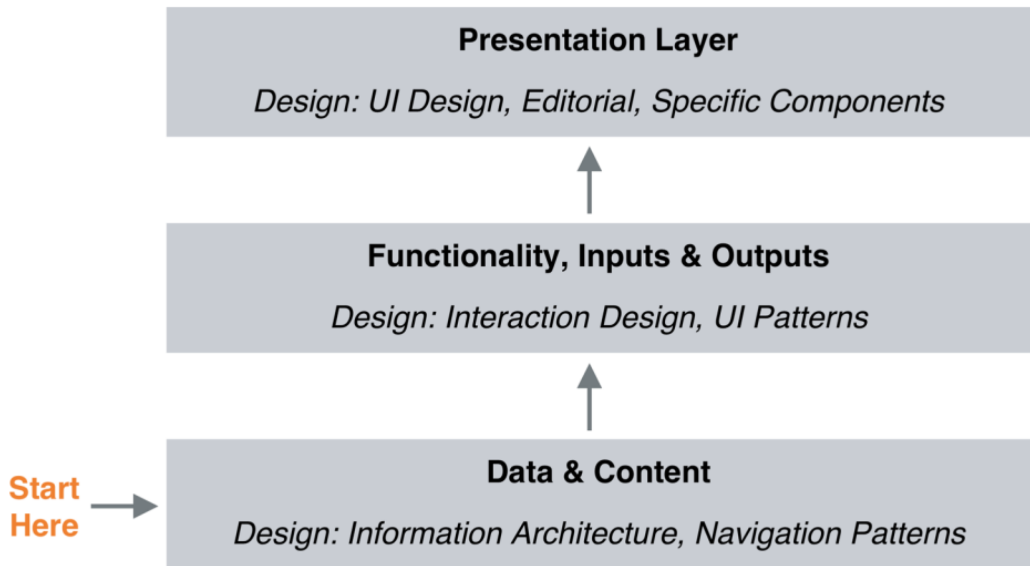


Figura 5.28: Las tres capas del diseño de interfaces gráficas

aparato, la retroalimentación de las entradas del piloto y las elecciones de parámetros para la simulación. A continuación, se presenta una lista no exhaustiva de ejemplos sobre la información presentada a fin de aclarar los requisitos, mostrándose su implementación en las secciones subsiguientes.

En primer lugar, la información sobre el estado de la simulación es necesaria para detectar situaciones de fallo y conocer los parámetros que se han configurado, por lo tanto, esto incluirá tanto el temporizador como posibles códigos de error dados. En segundo lugar, la información de vuelo incluye toda aquella información que proporciona el modelo matemático y se considera de interés para el usuario, esto incluye por ejemplo la posición y orientación del dron, las RPM de los motores, la velocidad de vuelo o el estado de las baterías. Asimismo, será de vital importancia comunicar al usuario el contexto en el que vuela el dron: alrededor de que se está volando y cómo afecta esto al vuelo. Por último, es importante que el piloto tenga información de lo que está haciendo, por lo que será necesario tanto representar los modos de vuelo activos mediante las entradas secundarias como la entrada primaria del estado del mando de control (ver Capítulo 4). A modo de resumen, la Tabla 5.1 sería toda la información que ha de intercambiarse con el usuario durante la simulación del vuelo.

En cuanto a la elección de parámetros necesarios de manera previa a la simulación, como puede ser la selección del dron, en cierto modo requiere un tratamiento separado por ser una parte de la experiencia de usuario desconectada de la visualización y simulación posterior. De igual

Estado de la simulación	Datos de vuelo	Entorno	Entradas del piloto
Tiempo de simulación	Posición	Elevación	Posición de los joysticks
Mensajes de error	Orientación	Obstáculos	Mando ON/OFF
Inicio/parada	Velocidad	Viento	Modos de vuelo
Avisos generados por el sistema	Altitud de vuelo	Colisiones	Funciones extra
	RPM de motores		
	% Batería		

Tabla 5.1: Ejemplos de posibles intercambios entre el usuario y la aplicación durante el vuelo

manera que la información "'previa al vuelo" requiere un tratamiento desconectado, en caso de implementar un sistema de misiones de entrenamiento, sería necesario así mismo incorporar información post vuelo sobre objetivos superados, como pueden ser *checkpoints* superados, trayectoria seguida o tiempo requerido para ejecutar ciertas maniobras.

Interacción con la simulación

Avanzando desde el primer paso de definición de requisitos, lo siguiente es plantear el otro requisito de la aplicación a desarrollar: que es añadir funcionalidades que permitan los intercambios de información requeridos, por ejemplo, modificar tanto en tiempo real como previo a comenzar la simulación diversos parámetros de la simulación, permitiendo así un amplio rango de circunstancias o perspectivas. Asimismo, se debe modelizar el tipo y la cantidad de opciones de interacción que se permite al usuario, para diseñar una experiencia que se sienta completa pero no confusa, por ejemplo, se saldría del alcance y propósito del proyecto dar control total al usuario sobre la aerodinámica del aparato, pero es interesante por ejemplo permitir al usuario elegir si volar con viento o no a fin de aprender el distinto comportamiento.

Aparte de la información que pasivamente se recibe a través de la interfaz gráfica como respuesta a la entrada primaria de control, también es importante determinar por tanto que otras funcionalidades se requieren, mas allá del pilotaje, y como pueden implementarse maneras de modificar parámetros de la simulación, tanto en tiempo real como de manera previa a la simulación.

En este caso, se contará con una selección de menús y botones interactuables que permitan las funcionalidades que se exponen mas adelante.

En el caso de modificaciones en tiempo real:

- Modificar la perspectiva
- Introducir fallos de motor o reiniciar estos motores
- Seleccionar modo manual/asistido
- Introducir la altitud para el modo Hover to Target
- Parar la simulación
- Arrancar la simulación
- Volver al menú principal para cambiar opciones

En cuanto a opciones pre-simulación:

- Seleccionar dron a volar (cuadricóptero con o sin gimbal, hexacóptero y octocóptero)
- Modificar el viento (fuerza, dirección y variabilidad)
- Batería disponible al inicio (%)
- Limitar imágenes por segundo (FPS)
- Modo ventana/Pantalla Completa

Un punto a tener en cuenta es como se interactúa con estas funcionalidades adicionales, en este proyecto se ha reforzado la separación entre entradas primarias (aquellas que controlan el dron) de las que afectan a la simulación pero no suponen parte de la "experiencia de pilotaje" mediante el soporte que permite esta interacción. Mientras que el dron se controla con un mando XInput genérico, para el resto de la interacción con la simulación se requiere el uso de teclado y ratón. Esta decisión se toma por simplificar y dirigir la experiencia, quitando esa parte del control de la simulación del foco principal.

En cuanto a las opciones gráficas de limitar la tasa de cuadros por segundo de la aplicación o la capacidad de ejecutar en modo venta escalable, aunque puedan parecer añadidos triviales son opciones esenciales para garantizar un mejor rendimiento en ordenadores con capacidades mas limitadas, pues hay que recordar que lo esencial es garantizar el rendimiento del modelo Simulink. Cabe destacar que en esta aplicación se mantendrá activada por defecto la opción de sincronización vertical, conocida como V-Sync, ya que esta limita las veces que se renderiza la imagen por segundo a la tasa de refresco del monitor. Esta opción presenta ciertas limitaciones

en cuanto a retardo de imagen, pero garantiza en nuestro caso que no se rendericen mas fotogramas de los necesarios limitando así el consumo de recursos del visualizador. Además, esta opción evitará problemas de lo que se conoce como “screen tearing”, que sucede cuando hay un cambio en el último fotograma generado por la GPU antes de que se pueda presentar el anterior completamente, generando una discontinuidad en la pantalla al haber renderizado medio fotograma en cada pasada.

5.2.2. Diseño de la interfaz

En cuanto al diseño de la interfaz del visor, se busca una solución de compromiso entre realismo y utilidad, asegurando de esta manera que disponemos de una interfaz intuitiva y útil. Es por esto que se incluyen dos tipos de información. En primer lugar, tendremos una serie de indicadores diegéticos que permitan representar cierta información de manera inmersiva, mientras que la mayoría de información se representa a través de distintos instrumentos simulados o indicadores de parámetros o variables de estado del dron que se proyectan sobre la visualización de la simulación, estando en primer plano para mayor accesibilidad por parte del usuario.

Asimismo, se diseña la experiencia de usuario de manera que se guíen de manera intuitiva las opciones disponibles al usuario, utilizando lenguaje sencillo y preciso y elementos visuales claros.

Menú principal

En primer lugar, la aplicación empieza en un menú desde el que configurar los parámetros antes de comenzar lanzar el visor y lanzar la simulación (Figura 5.29).

Se opta por un diseño y distribución sencillas, con tan solo tres opciones: comenzar, opciones o salir. En el menú de opciones podremos seleccionar los parámetros antes mencionados, estos se guardarán en el Registro de Windows mediante las “PlayerPrefs” que nos permite usar Unity, de esta manera las selecciones se guardarán, incluso entre sesiones distintas. Cabe destacar que esta información también será accesible al modelo Simulink, haciendo uso de la herramienta correspondiente desarrollada por Y. Altman [29] para poder determinar el dron seleccionado u otras opciones.

Información diegética

La información diegética se refiere a aquella que se genera como parte del entorno simulado, es decir, aquello que vería u oiría el piloto y que percibirá el usuario. La mayoría de estas han sido descritas anteriormente cuando se expusieron los distintos elementos del entorno y los efectos

Figura 5.29: Menú principal

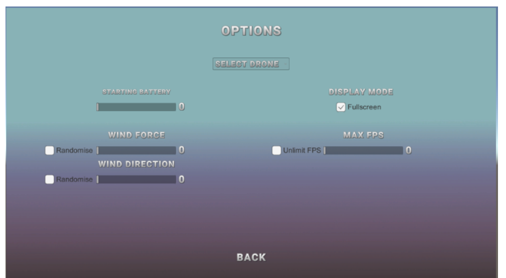


Figura 5.30: Menú de opciones

que se le aplican e incluyen por ejemplo:

- Indicaciones visuales del efecto del viento en el entorno, por ejemplo, el movimiento de árboles, partículas o mangas de viento en función de la fuerza y dirección del mismo.
- La modificación del sonido de los rotores en función de las RPM de los motores da indicación auditiva del estado de los motores, además, se implementa una función nativa de Unity para simular el efecto de la distancia en el sonido.
- Renderizados de sombras dinámicos, que darían una idea de la altitud de vuelo del dron al ver la proyección de su sombra sobre el entorno.

Información extradiegética

Esta es la información que existe fuera del entorno simulado, pero es necesaria para el correcto entendimiento de lo que está sucediendo, así como para interactuar con el modelo. Se implementa a través de una serie de instrumentos e indicadores que se presentan en una capa superpuesta al resto de la visualización.

Tomando como base la división de información requerida en categorías establecida en el Apartado 5.2.1, se diseñará la interfaz de manera que agregue en distintos grupos los indicadores necesarios. Asimismo, la naturaleza de cada dato se prestará a que sea mas conveniente representarlo de una u otra manera, y aquí también entrará una jerarquía en cuando a importancia y precisión requerida de cada dato. La manera en la que se conecta la información en cada una de estas agrupaciones también influirá en como se percibe la misma, será por lo tanto vital seguir principios de diseño de interfaces en cuanto a su distribución y presentación, como pueden ser los de la psicología de Gestalt y el estudio del flujo visual entre los elementos [28].

Por ejemplo, cabría preguntarse ¿de qué manera necesita conocer el estado de carga que queda en las baterías un piloto? El usuario estará quizás mas interesado en obtener esta infor-

mación de manera cualitativa, por ejemplo, sabiendo que le queda poca o mucha batería, que en conocer el valor actual en MAh de la carga en cada una de las dos baterías, ya que está información no solo sería innecesaria, si no que requiere de conocimiento previo para su interpretación como cuál es la capacidad de la batería para cada uno de los drones. Siguiendo con este ejemplo, si tan solo lo representáramos como un número en una esquina, se perdería entre los demás elementos y no se percibiría como importante, sin embargo, si lo representamos de manera visual como una barra que comienza verde y hace una transición al rojo a medida que se descargan las baterías, podemos generar un interés visual para el usuario de manera que obtenga la información necesaria sin distraer demasiado.

Prototipos

Basándose en estos principios, se busca en primer lugar la agrupación de estas funcionalidades e información a mostrar en conjuntos lógicos que permitan guiar. Es importante en este paso reflexionar sobre la cantidad y forma en la que se presenta la información a fin de no abrumar o distraer al usuario con excesivo ruido visual como hemos hablado. A este fin, se fueron realizando diversos prototipos (Figura 5.33), con las siguientes directrices:

- En primer lugar, se mantendrá el dron como el foco en todo momento, por lo que los elementos de la interfaz no han de obstaculizar su visión en ningún momento a ser posible, tanto con la cámara fija como la cámara de seguimiento en tercera persona. Por lo tanto y en la medida de lo posible, los distintos elementos se limitarán a los bordes de la pantalla.

- Se decide distribuir la información a transmitir en los siguientes 4 grandes grupos: datos de la simulación, datos de vuelo, datos de entrada y opciones interactuables.

En base a eso, y los criterios anteriormente mencionados, los prototipos avanzaron llegando a las siguientes conclusiones: en primer lugar, cabe destacar que los que se consideran mas importantes, y los que deben por tanto tener prioridad durante el vuelo, serán los datos de vuelo y entrada. Estos deben ser accesibles incluso en mitad del vuelo con tan solo un vistazo, por lo que se optará en ese caso por objetos visuales que transmitan la información de manera cualitativa. En cuanto a los datos de simulación, para ellos será importante representarlos cierta precisión, pero pueden estar relegados a un segundo plano en la jerarquía al ser información que será importante sobre todo en caso de fallo de la aplicación o simulación para diagnosticar el error. Por último, pero no menos importante, tenemos las opciones interactuables, aquí se busca una solución que sea minimalista a fin de distraer lo mínimo, pero que ofrezca toda la fun-

cionalidad requerida. Por ello, se opta por una serie de botones organizados para las opciones que requieren acceso mas inmediato, mientras que el resto se desplegarán mediante un menú emergente de funciones adicionales.

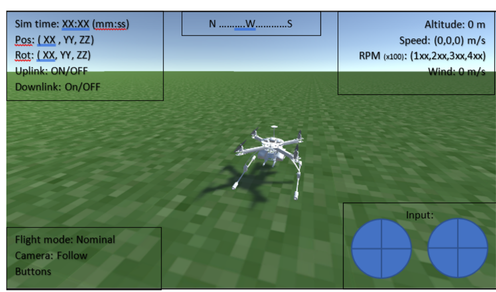


Figura 5.31: Prototipo Inicial

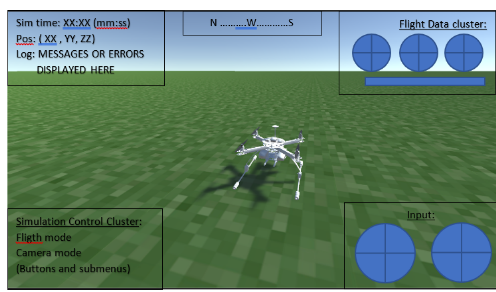


Figura 5.32: Prototipo Final

Figura 5.33: Prototipos de interfaz

Diseño final

Con todo lo anterior, finalmente llegamos al diseño de la interfaz definitivo, que puede verse en la Figura 5.34. Esta proporciona una interfaz minimalista pero que aglutina los elementos visuales al lado derecho de la pantalla, mientras que los grupos de interfaz del estado de la simulación y el control de la misma se queda a la izquierda, de esta manera se agrupan los grupos según su utilidad y relación entre ellos. A continuación se expone brevemente cada uno de estos elementos, que pueden verse con mayor detalle en la Figura 5.36.

Figura 5.34: Diseño Final de la interfaz

Todos estos elementos se organizan bajo un solo “GameObject” tipo “UI Canvas”, que como su nombre indica será el lienzo sobre el que se renderizan los elementos de IU, para poder adaptar la disposición y tamaño de los elementos en función de la resolución de ejecución de

la aplicación, manteniendo la distribución escogida. De esta manera, se asegura una buena representación en mayor variedad de dispositivos.

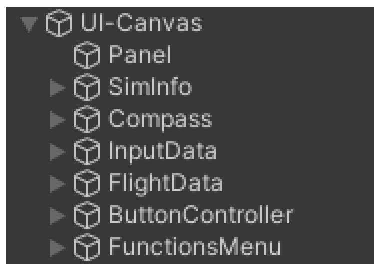
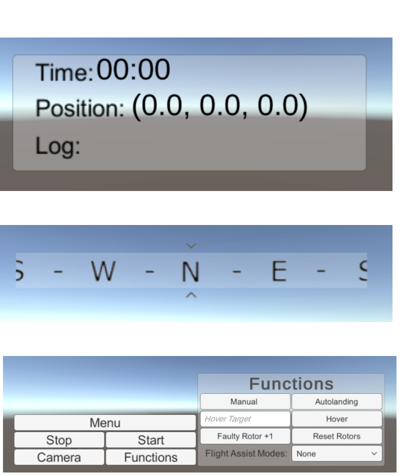
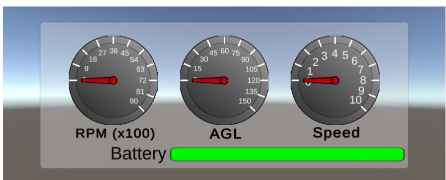


Figura 5.35: Componentes del GameObject UI Canvas

Por tanto, en cuanto a los elementos finales de la interfaz gráfica diseñada tenemos, de izquierda a derecha y de arriba a abajo en la interfaz:

- **Datos de la simulación:** En la esquina superior izquierda se incorpora un pequeño panel semitransparente que mostrará tan solo texto, incluyendo el tiempo, la posición actual del dron y un espacio para mensajes de aviso y de error necesarios.
- **Brújula:** se añade como complemento a la posición de los datos la orientación cardinal del dron en forma de brújula móvil.
- **Datos de vuelo:** se presentan en tres relojes las RPM medias, la altura sobre el suelo y la velocidad de vuelo. Así mismo, se muestra en una barra el estado actual de carga de la batería, este comienza en color verde e irá al rojo cuando quede poca batería.
- **Datos de entrada:** con un sencillo gráfico con forma de mando remoto, este indicador muestra el estado de conexión con el mando y la posición actual de los sticks.
- **Botonera:** Se concentran todas las opciones interactuables en una botonera simple y pequeña a fin de que no distraiga, las opciones de funciones adicionales están a priori ocultas en un submenú desplegable.





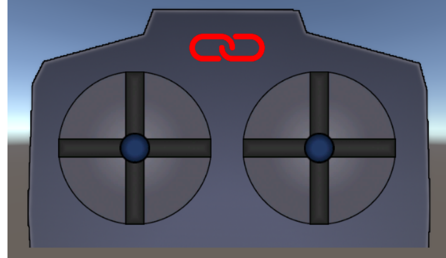


Figura 5.36: Detalle de elementos de la interfaz

Capítulo 6

Resultados

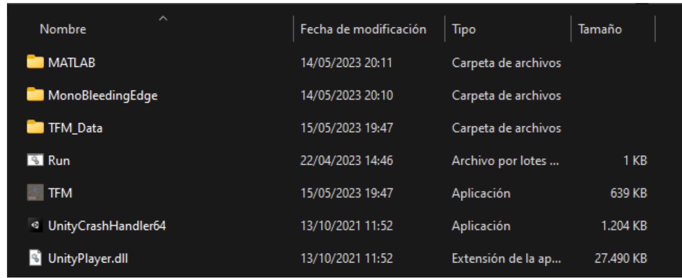
En este capítulo se describirán brevemente los resultados obtenidos en este proyecto. En primer lugar, se introduce el producto final: la aplicación generada por Unity. Asimismo, se detalla el proceso de instalación de los ficheros necesarios de Matlab para el correcto funcionamiento, así como los requisitos para ejecutar el simulador y el hardware para el que está validado. Adicionalmente, se evalúa la respuesta general del sistema SIL, denotando las limitaciones existentes.

6.1. Instalación y requisitos

En cuanto a la instalación, al compilar la aplicación de Unity, se genera una aplicación ejecutable de Windows así como carpetas con todos los archivos necesarios comprimidos de manera optimizada. La única acción requerida al usuario una vez se cuenta con la carpeta autogenerada por Unity es colocar la carpeta de MATLAB con el modelo, los constructores y demás código de soporte, así como la interfaz en forma del archivo "$Run.bat$" que permite hacer la llamada desde la aplicación a MATLAB para ejecutar los constructores correspondientes y la simulación dentro de la carpeta en la que se encuentra el ejecutable, el resultado se puede observar en la Figura 6.1. Este proceso puede automatizarse produciendo un instalador, como por ejemplo mediante la herramienta gratuita Inno Setup, de esta manera se comprimen los archivos y permite una experiencia mas directa para el usuario, pudiendo compartir la totalidad del proyecto con un solo fichero de instalación.

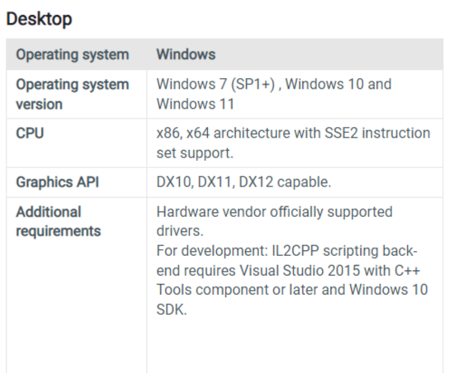
Esta aplicación ha sido desarrollada utilizando MATLAB versión R2022b Update 2 y Unity Editor 2020.3.21f1, a modo de referencia se indican en las siguientes Figuras 6.2 y 6.3 los requisitos mínimos para ejecutar el software requerido. En cuanto al espacio requerido en disco, el tamaño

del instalador es de tan solo 88,2MB, mientras que el tamaño final de la carpeta es de 283MB, de los cuales 70,8MB corresponden a la carpeta de MATLAB.



Nombre	Fecha de modificación	Tipo	Tamaño
MATLAB	14/05/2023 20:11	Carpeta de archivos	
MonoBleedingEdge	14/05/2023 20:10	Carpeta de archivos	
TFM_Data	15/05/2023 19:47	Carpeta de archivos	
Run	22/04/2023 14:46	Archivo por lotes ...	1 KB
TFM	15/05/2023 19:47	Aplicación	639 KB
UnityCrashHandler64	13/10/2021 11:52	Aplicación	1.204 KB
UnityPlayer.dll	13/10/2021 11:52	Extensión de la ap...	27.490 KB

Figura 6.1: Estructura de la carpeta de la aplicación para su funcionamiento



Desktop

Operating system	Windows
Operating system version	Windows 7 (SP1+) , Windows 10 and Windows 11
CPU	x86, x64 architecture with SSE2 instruction set support.
Graphics API	DX10, DX11, DX12 capable.
Additional requirements	Hardware vendor officially supported drivers. For development: IL2CPP scripting back-end requires Visual Studio 2015 with C++ Tools component or later and Windows 10 SDK.

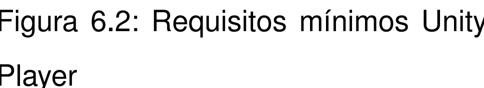
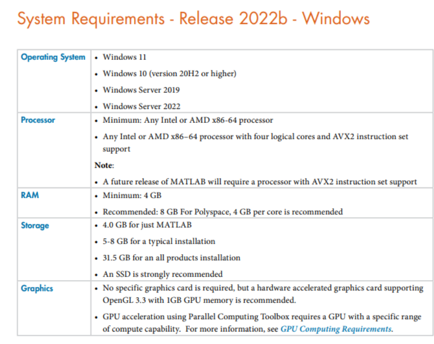


Figura 6.2: Requisitos mínimos Unity Player



System Requirements - Release 2022b - Windows

Operating System	• Windows 11 • Windows 10 (version 20H2 or higher) • Windows Server 2019 • Windows Server 2022
Processor	• Minimum: Any Intel or AMD x86-64 processor • Any Intel or AMD x86–64 processor with four logical cores and AVX2 instruction set support **Note**: • A future release of MATLAB will require a processor with AVX2 instruction set support
RAM	• Minimum: 4 GB • Recommended: 8 GB For Polyspace, 4 GB per core is recommended
Storage	• 4.0 GB for just MATLAB • 5-8 GB for a typical installation • 31.5 GB for an all products installation • An SSD is strongly recommended
Graphics	• No specific graphics card is required, but a hardware accelerated graphics card supporting OpenGL 3.3 with 1GB GPU memory is recommended. • GPU acceleration using Parallel Computing Toolbox requires a GPU with a specific range of compute capability. For more information, see *GPU Computing Requirements*.

Figura 6.3: Requisitos MATLAB R2022b

6.2. Limitaciones

Como parte de los resultados, es importante analizar los problemas que se han detectado durante el estudio de este proyecto, como se han mitigado o teorizar si podrían con desarrollos posteriores solucionarse.

En cuanto a problemas conocidos y limitaciones encontradas relativas a la implementación elegida, se pueden destacar las siguientes:

- Velocidades de vuelo elevadas cuando se acerca al terreno u obstáculos que suponen que el dron atraviesa dicho suelo. Esto se ha mitigado mediante un ajuste a la condición suelo, sin embargo, la limitación parece estar dada por la combinación del tiempo de cálculo de físicas en Unity para detectar las colisiones, el retardo en el enlace de subida UDP (que informa del AGL actual) y el tiempo de iteración del modelo Simulink. Todo esto supone

que, cuando estamos en un terreno elevado (es decir, el suelo no se encuentra en z=0) se puede llegar tener una altura negativa si la velocidad del dron hacia el suelo es tal que en el paso de una iteración a la siguiente se atraviesa el suelo sin haber detectado una colisión. Es posible que una implementación distinta, con mayor cálculo de físicas en Unity, pudiera solucionar este problema, pero sin embargo se considera que esta solución sería contraria a la filosofía e intenciones del proyecto de generar un visualizador para el modelo Simulink limitando el impacto de cálculo del visor.

- Ejecutar el modelo Simulink desde Unity es funcional, sin embargo, debido al funcionamiento de Windows y a la necesidad de ejecutar la ventana de comandos de Matlab, nos minimizará la aplicación cuando se ejecute, teniendo que volver al visor una vez se lanza el código Matlab. Añadido a esto, en la primera sesión del día, el tiempo para compilar el modelo y generar los datos de caché es elevado. En este sentido, se ha mantenido como opción el lanzar manualmente el modelo desde la interfaz gráfica de Simulink habitual, manteniendo la funcionalidad con menos comodidad o inmersión pero algo mas de agilidad.

- El uso de ángulos de Euler para realizar los cálculos de actitud del aparato tienen el potencial problema de la indeterminación de la guiñada, es por eso que Unity por ejemplo define las rotaciones mediante cuaterniones. En este caso, los drones utilizados no están previstos para hacer maniobras tan agresivas que los llevaran a tener un ángulo de cabeceo o balanceo de 90º, lo que llevaría al "*gimbal-lock*", por lo que aunque no es un problema, sería conveniente y mas cómodo para la implementación en Unity una adaptación del modelo dinámico para trabajar con cuaterniones.

6.3. Demostración

Si bien es cierto que es una aplicación ligera, esta no ha sido validada para una gama de ordenadores de características diversas, por lo que el rendimiento aquí estudiado podría diferir según la máquina utilizada y no se pueden determinar requisitos recomendados. En cualquier caso, se ha comprobado y validado el rendimiento para un ordenador de sobremesa de gama media, con procesador AMD Ryzen de 6 núcleos, 16GB de RAM DDDR4 y una tarjeta gráfica con 6GB GDDR6 de memoria de vídeo.

Centrándose en la maquina mencionada, se realizan una serie de *benchmarks* para medir el rendimiento, tanto en consumo de recursos, como en cuanto a tasa de fotogramas por segundo.

Asimismo, se da una valoración cualitativa de la respuesta del sistema. Destacar en este punto que en Unity se ha configurado la opción de sincronización vertical (conocido también como V-Sync) y que se ha utilizado un monitor de 144Hz de tasa de refresco, por lo que, de funcionar sin impedimento, la aplicación debería limitarse alrededor de los 144fps. Se muestra en la siguiente Tabla 6.1 por lo tanto el resumen de los benchmarks, habiéndose realizado varios ensayos de 60 segundos para cada dron, utilizando tanto la cámara fija (30 segundos) como la cámara en tercera persona (30 segundos) y realizando diversas maniobras en lo que podría ser un “uso normal” de la aplicación.

FPS	Quad	Hexa	Octo
Mínimo	127	142	142
Máximo	145	145	145
Media	143,68	143,98	143,98

Tabla 6.1: Tasas de fotogramas (FPS) de la aplicación (V-Sync activado @144Hz)

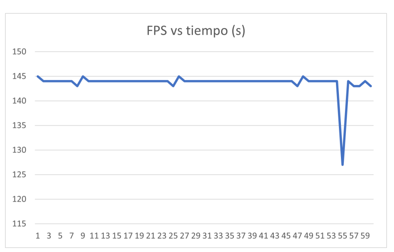


Figura 6.4: Cuadricóptero

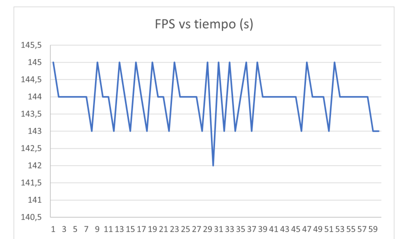


Figura 6.5: Hexacóptero

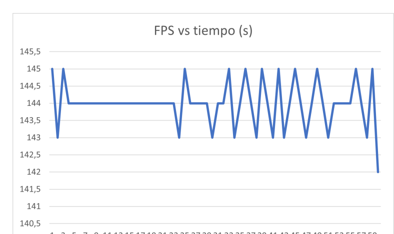
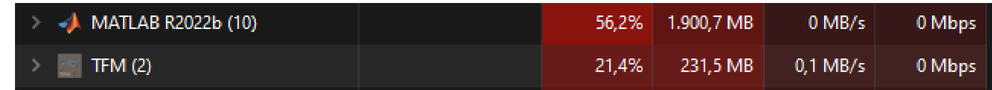


Figura 6.6: Octocóptero

Como puede observarse en las Figuras 6.4, 6.5 y 6.6 correspondientes al primer ensayo, se mantiene sin ningún problema la media requerida por la aplicación. Tan solo hubo una muestra de datos anómala en todos los ensayos, en este caso en el Quad, pero no se percibió en ningún momento en cuanto a a la experiencia visual se refiere. Cualitativamente se puede decir que la simulación parece correr a tiempo real satisfactoriamente, con poco retardo de entrada (la respuesta al mando se siente inmediata).

En cuanto al consumo de recursos de las distintas partes del entrenador, podemos ver en la Figura 6.7 que, en un caso típico de vuelo rectilíneo uniforme durante uno de los ensayos, el consumo de memoria RAM y utilización de la CPU es bastante limitado por parte de la aplicación de visualización, lo que permitirá un buen rendimiento a la simulación de MATLAB, cumpliendo así uno de los objetivos establecidos.



MATLAB R2022b (10)	56,2%	1.900,7 MB	0 MB/s	0 Mbps
TFM (2)	21,4%	231,5 MB	0,1 MB/s	0 Mbps

Figura 6.7: Consumo de recursos nominal Aplicación vs Modelo MATLAB

Capítulo 7

Conclusiones

La importancia del sector de drones es palpable en la actualidad debido a su desarrollo tecnológico e industrial reciente, y en el marco de nuevas normativas de fabricación y operación, la necesidad de disponer de software que facilite tanto familiarizarse como entrenarse en el manejo de estos aparatos es cada vez mas importante para garantizar la seguridad operacional. En concreto, debido a su fácil accesibilidad y gran variedad de drones disponibles, es importante que estas herramientas sean modulares, sencillas y cómodas, por lo que el diseño que se ha realizado permitiría fácilmente expandirse y añadir más opciones en caso de nuevas necesidades, como puedan ser distintos modelos de dron o nuevas funcionalidades.

A lo largo de este proyecto se ha desarrollado un entrenador de vuelo para aeronaves RPAS multirrotor con configuraciones de 4, 6 y 8 rotores y un visualizador independiente del modelo que permite en tiempo real interactuar con la simulación y proporciona una interfaz gráfica para el modelo Simulink. Se puede concluir, dados los resultados, que Simulink es una herramienta potente y capaz para simulación de RPAS multirrotor en tiempo real, ofreciendo comodidades en el desarrollo y capacidad de cálculo suficiente para realizar una simulación realista. Por otro lado, el entorno de desarrollo de Unity ha supuesto grandes facilidades a la hora de desarrollar una aplicación de visualización e interacción, tanto por las herramientas nativas como por la documentación y ejemplos disponibles debido a que tiene una gran comunidad colaborativa.

7.1. Trabajo futuro

A continuación se presenta una lista de las posibles características a implementar sobre el desarrollo actual del entrenador:

- Añadir funciones de autopiloto completo, pudiendo añadir funcionalidad RTH realimentando la posición inicial o una posición indicada.
- Implementar una mejora al modelo para reaccionar ante colisiones laterales diferenciado del actual sistema para detectar colisiones con el suelo. Esto conllevaría asimismo a simular impactos en las hélices que pudieran inutilizar motores, así como la necesidad de implementar en el modelo dinámico la posibilidad de implementar fuerzas de reacción ante estas colisiones.
- Implementar un sistema de misiones de entrenamiento, con ejercicios de destreza para perfeccionar las capacidades del usuario en el vuelo de drones. Para ello habría que definir una serie de "*checkpoints*" por los que el dron tuviera que pasar en orden e ir guardando el estado actual, así como posiblemente la trayectoria seguida y el tiempo tomado.
- Implementar nuevas indicaciones respecto a normativa vigente, así escenarios con *geo-fencing* para prevenir entrar en zonas restringidas o prohibidas y reaccionar si se entra, así como escenarios con *geo-caging* para operaciones limitadas a un espacio aéreo (por ejemplo, un club de vuelo). Esto podría ir ligado al autopiloto completo anteriormente mencionado, asegurando que el dron dispone de un mecanismo automático para salir de zonas restringidas o aterrizar automáticamente.

Bibliografía

[1] G. Linares, "Design and Integration a Software in the Loop for a Fixed Wing RPA," TFM, Universidad Carlos III de Madrid, 2021. Tutor: Federico Martín de la Escalera.

[2] I. Timmermans, "Entrenador basado en modelo parametrizado para pilotos de multirrotor," TFG, ETSII, Universidad Politécnica de Madrid, 2020. Tutor profesional: Federico Martín de la Escalera.

[3] A. Prol Fernández, "Desarrollo de un modelo software in the loop (sitl) de una plataforma no tripulada," TFG, ETSIAE, Universidad Politécnica de Madrid, 2022. Tutor profesional: Federico Martín de la Escalera.

[4] A. Fernández Guerrero, "Avances en el desarrollo de un simulador de RPAS integrando vehículo hexacóptero y octocóptero," TFM, Universidad Europea de Madrid, 2023. Tutor: Federico Martín de la Escalera.

[5] AESA, "Libro blanco de I+D+i para la aviación no tripulada en españa," white paper, AESA, Diciembre 2022.

[6] D. Falanga, K. Kleber, S. Mintchev, D. Floreano, and D. Scaramuzza, "The foldable drone: A morphing quadrotor that can squeeze and fly," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 209–216, 2019.

[7] K. Rawlinson, "Drone hits plane at heathrow airport, says pilot," *The Guardian*, 2016. https://www.theguardian.com/uk-news/2016/apr/17/drone-plane-heathrow-airport-british-airways.

[8] EASA, "Advance notice of proposed amendment 2015-10: Introduction of a regulatory framework for the operation of drones," 2015. https://www.easa.europa.eu/en/document-library/notices-of-proposed-amendment/npa-2015-10.

[9] European Commission, "Commision Delegated Regulation (EU) 2019/945," 2019. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32019R0945.

[10] European Commission, "Commision Delegated Regulation (EU) 2019/947," 2019. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32019R0947.

[11] AESA, "Normativa europea de UAS/drones," 2022. https://www.seguridadaerea.gob.es/es/ambitos/drones/normativa-europea-de-uas-drones.

[12] K. P. Valavanis and G. J. Vachtsevanos, *Handbook of Unmanned Aerial Vehicles*. Springer Dordrecht, 5 ed., 2015.

[13] FAA, US Department of Transportation, *Rotorcraft Flying Handbook*, 2000. FAA-H-8083-21.

[14] P. Sanchez-Cuevas, G. Heredia, and A. Ollero, "Characterization of the aerodynamic ground effect and its influence in multirotor control," *International Journal of Aerospace Engineering*, vol. 2017, p. 1823056, Aug 2017.

[15] G. Hattenberger, M. Bronz, and J.-P. Condomines, "Evaluation of drag coefficient for a quadrotor model," *International Journal of Micro Air Vehicles*, vol. 15, p. 17568293221148378, 2023.

[16] K. Ogata, *Ingeniería de control moderna*. Pearson Educación, 2003.

[17] V. M. Martínez, F. M. de la Escalera, and S. G. Ruiz, *Curso para pilotos de RPAS: enfoque práctico*. Víctor Mateo Martínez, 5 ed., 2016.

[18] R. Sagrén and A. Yun, "Quadrotor in y4 configuration," Master's thesis, KTH ROYAL INSTITUTE OF TECHNOLOGY, 2017.

[19] A. Nemati and M. Kumar, "Modeling and control of a single axis tilting quadcopter," in *2014 American Control Conference*, pp. 3077–3082, 2014.

[20] P. D. Groves, *Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems*. Norwood, Massachusetts, United States of America: Artech House, 2008.

[21] R. García González, "Modelado y calibracion de sensores inerciales MEMS para su uso como AHRS," TFG, ETSIAE, Universidad Politécnica de Madrid, 2019.

[22] M. Voigt, "Simulink Xbox Controller (XInput API)," 2023. https://github.com/MatVo1992/Simulink-XInput-Controller, GitHub. Consultado 15 de Febrero, 2023.

[23] J. Postel, "User datagram protocol," RFC 768, RFC Editor, Agosto 1980.

[24] C. Doğan, "Example udp receiver and transmitter project in unity," 2019. https://github.com/CihadDogan/UDPExample/tree/master#readme, GitHub. Consultado 10 de Febrero, 2023.

[25] ALP, "Asset package: Grass flowers pack free," 2019. https://assetstore.unity.com/packages/2d/textures-materials/nature/grass-flowers-pack-free-138810, Unity Asset Store. Consultado 15 de Mayo, 2023.

[26] Pixel Games, "Asset package: Realistic tree 9 [rainbow tree]," 2016. https://assetstore.unity.com/packages/3d/vegetation/trees/realistic-tree-9-rainbow-tree-54622, Unity Asset Store. Consultado 12 de Febrero, 2023.

[27] D. Norman, *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books, 2013.

[28] J. Tidwell, C. Brewer, and A. Valencia, *Designing Interfaces*. O'Reilly Media, Inc, 3 ed., 2020.

[29] Y. Altman, "Matlab Registry Utilities," 2023. https://es.mathworks.com/matlabcentral/fileexchange/73045-windows-registry-utilities, MATLAB Central File Exchange. Consultado 24 de Abril, 2023.

[30] A. Fernández López, "Adaptación de controlador de dron de ala fija a multirrotor e integración en simulador de vuelo," TFM, Universidad Europea de Madrid, 2022. Tutor: Federico Martín de la Escalera.

Anexo A

Código

A.1. Código Matlab

A.1.1. Constructores

Estos códigos permiten inicializar todas las variables necesarias para cada caso implementado. Asimismo, proporcionan flexibilidad al modelo al poder ser modificados independientemente del mismo para añadir nuevos drones o cambiar sus características físicas. Pueden consultarse en el trabajo de Alejandro Fernández [4].

A.1.2. Código de lanzamiento de simulación

Este código es el que se llama desde Unity y selecciona el dron activo para lanzar el constructor correspondiente antes de ejecutar el modelo Simulink. Este es un desarrollo del código de [1] y [30].

```
%% Created by Raul Garcia y Alejandro Fernandez as part of their final Master
    's Theses.
% Based on code by Guillermo Linares and Arturo Fernandez

global dron
close all
clear all
clc

%% Sampling Speed
dt = 1/400;
```

```
11
12 %% Drone Selection
13
14 % Unity stores drone selection in Key Registry, this code reads it and will
15 %prompt user for input if missing
16
17 DronSelected = getRegistryValue('HKEY_CURRENT_USER\Software\RGG\TFM','
     DroneSelected_h2248507710');
18
19 if DronSelected == '515541445F574700'
20     i=41;
21 else if DronSelected == '515541445f574f4700'
22     i=42;
23 else if DronSelected == '4845584100'
24     i=6;
25 else if DronSelected == '4f43544f00'
26     i=8;
27 end
28
29 %% Run Constructor
30
31 cond = 0;
32
33 while cond == 0
34     if i == 41
35         disp('Quadcopter with gimbal');
36         dron = constructor_WG(4,45);
37
38         K_Thrust     = 25;
39         D_Thrust     = 0.8;
40         P_Thrust     = 2;
41
42         K_Yaw        = 1.5;
43         P_Yaw        = 2;
44
45         K_Roll       = 20;
46         Pl_Roll      = 0.022;
47         P_Roll       = 1;
```

```
48          D_Roll       = 0.4;
49
50          K_Pitch      = 20;
51          Pl_Pitch     = 0.022;
52          P_Pitch      = 1;
53          D_Pitch      = 0.4;
54          cond = 1;
55
56      elseif i == 42
57          disp('Quadcopter without gimbal');
58          dron = constructor_WOG(4,45);
59
60          K_Thrust     = 23;
61          D_Thrust     = 0.8;
62          P_Thrust     = 2;
63
64          K_Yaw        = 1.5;
65          P_Yaw        = 2;
66
67          K_Roll       = 20;
68          Pl_Roll      = 0.022;
69          P_Roll       = 1;
70          D_Roll       = 0.4;
71
72          K_Pitch      = 20;
73          Pl_Pitch     = 0.022;
74          P_Pitch      = 1;
75          D_Pitch      = 0.4;
76
77          cond = 1;
78
79      elseif i == 6
80
81          disp('Hexacopter without gimbal');
82          dron = constructor_Hexa(6,30);
83
84          K_Thrust     = 28;
85          D_Thrust     = 0.8;
```

```
86          P_Thrust     = 2.5;
87
88          K_Yaw        = 1.3;
89          P_Yaw        = 1.5;
90
91          K_Roll       = 20;
92          Pl_Roll      = 0.023;
93          P_Roll       = 1;
94          D_Roll       = 0.5;
95
96          K_Pitch      = 20;
97          Pl_Pitch     = 0.023;
98          P_Pitch      = 1;
99          D_Pitch      = 0.5;
100
101         cond = 1;
102
103     elseif i == 8
104
105         disp('Octocopter without gimbal');
106         dron = constructor_Octo(8,0);
107
108         K_Thrust     = 40;
109         D_Thrust     = 0.8;
110         P_Thrust     = 2;
111
112         K_Yaw        = 1.5;
113         P_Yaw        = 2;
114
115         K_Roll       = 20;
116         Pl_Roll      = 0.022;
117         P_Roll       = 1;
118         D_Roll       = 0.4;
119
120         K_Pitch      = 20;
121         Pl_Pitch     = 0.022;
122         P_Pitch      = 1;
123         D_Pitch      = 0.4;
```

```
124
125         cond = 1;
126
127
128     else
129         disp('Selection error. Select drone model:')
130         disp('"41" for Quadcopter with gimbal')
131         disp('"42" for Quadcopter without gimbal')
132         disp('"6" for Hexacopter without gimbal')
133         disp('"8" for Octocopter without gimbal')
134         i = input(' ');
135
136     end
137 end
138 %% RUN SIMULINK
139 sim('RunCase_Multirrotor_SL.slx')
```

A.1.3. Selección de modo

Este sencillo código tan solo establece el valor de las distintas *flags* utilizadas para los modos en base a la variable ”funcionesrecibida por UDP.

```
1  function [RLMode, FBMode, controlmode] = ModeSelect(funciones)
2  RLMode = 0;
3  FBMode = 0;
4  controlmode = 0;
5  if funciones == 0
6      controlmode = 0;
7      RLMode = 0;
8      FBMode = 0;
9  end
10 if funciones == 1
11     controlmode = 0;
12     RLMode = 0;
13     FBMode = 1;
14 end
15 if funciones ==2
16     controlmode = 0;
17     RLMode = 0;
```

```
18      FBMode = -1;
19  end
20  if funciones ==3
21      controlmode = 0;
22      RLMode = 1;
23      FBMode = 0;
24  end
25  if funciones ==4
26      controlmode = 0;
27      RLMode = -1;
28      FBMode = 0;
29  end
30  if funciones ==5
31      controlmode = 1;
32      RLMode = 0;
33      FBMode = 0;
34  end
35  end
```

A.1.4. Condición suelo

Este código determina si el dron está tocando el suelo o no, a fin de establecer las limitaciones de movimiento correspondientes. Por una parte, toma la medida de *Weight On Wheels* de Unity, pero debido a que a altas velocidades puede llegar a atravesar el suelo sin registrar la colisión, se incluyen también las medidad de AGL y elevación para que, en casos extremos, no siga bajando. Este código es realimentado en el control del modelo, asumiendo que una vez estás posado en el suelo, el único mando posible es el eje de thrust hacía arriba para alejarse del mismo.

```
1  function height  = Height(WOW, AGL,z)
2
3  if WOW == 1 || AGL<(0.01) || z>0
4      height = 0;
5  else
6      height = 1;
7  end
8  end
```

A.2. Código Unity (C #)

A.2.1. UpdatePos.cs

Este código actualiza la posición y actitud del dron en base a la información recibida por el enlace UDP. Asimismo, este es el código en el que se mide el AGL, proyectando un “rayo” desde el dron hasta el primer obstáculo (o el suelo) y se comprueba el WoW y las colisiones laterales.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class UpdatePos : MonoBehaviour
{
    public UDPReceiver Receptor;
    public Rigidbody dron;
    public Collider GroundDetector;
    public Collision SidesDetector;
    private Vector3 globalDown;
    RaycastHit hit ;
    public float heightAboveGround;
    public bool groundCollision;
    public Vector3 collisionPos;

    void Start()
    {
        globalDown = new Vector3(0.0f, -1.0f, 0.0f);
        groundCollision = false;
    }

    void Update()
    {
        if (Physics.Raycast(dron.transform.position, globalDown, out hit)) //
            Esta parte del codigo es la que mide el AGL
        {
            heightAboveGround = hit.distance;
        }
        transform.position = new Vector3((float) Receptor.simX , (float)
            Receptor.simY , (float)Receptor.simZ);
```

```
            Vector3 rotato = new Vector3((float)Receptor.simroll , (float)
                Receptor.simyaw , (float)Receptor.simpitch );
            transform.rotation = Quaternion.Euler(rotato);

             if (heightAboveGround > 120)
            {
                Debug.Log("<color=red>" + "Altura maxima excedida" + "</color>");
            }
        }

        private void OnTriggerEnter(Collider GroundDetector)
        {
            groundCollision = true;
        }

        private void OnTriggerExit(Collider GroundDetector)
        {
            groundCollision = false;
        }

        private void OnCollisionEnter(Collision SidesDetector)
        {
            ContactPoint contact = SidesDetector.contacts[0];
            collisionPos = contact.point;
        }
}
```

A.2.2. CommunicationController.cs

Este código maneja las comunicaciones haciendo uso de los scripts UDPReceiver y UDPTransmitter como se ha descrito en el capitulo 4 y 5.

```
using UnityEngine;
using System;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Diagnostics;

```

```
8
9 public class CommunicationController : MonoBehaviour, IReceiverObserver
10 {
11     UDPReceiver _UdpReceiver;
12     UDPTransmitter _Udpransmitter;
13
14     //Objetos de los que leer
15     public ButtonUI Botonera;
16     public FunctionsButtonUI FunctionsBotonera;
17     public UpdatePos quadcopter;
18     public UpdatePos quadcopterWOG;
19     public UpdatePos hexacopter;
20     public UpdatePos octocopter;
21     private UpdatePos ActiveDrone;
22     public WindControl Wind;
23     private string DroneSelected;
24
25     // Info a enviar (enlace de subida)
26     public double simSTOP;
27     public double simAGL;
28     public double simWOW;
29     public double WindDirection;
30     public double WindForce;
31     public double nfallos;
32     public double RotorFallido;
33     public double FunctionSelected;
34     public double HoverTarget;
35     public double AutolandingMode;
36     public double[] UpValues;
37     public double[] values;
38
39     //Info a recibir (enlace de bajada)
40     public double simtime;
41     public double simX;
42     public double simY;
43     public double simZ;
44     public double simroll;
45     public double simpitch;
```

```
46      public double simyaw;
47      public Vector3 simspeed;
48      public double SoC;
49      public double RPM1;
50      public double RPM2;
51      public double RPM3;
52      public double RPM4;
53      public double RPM5;
54      public double RPM6;
55      public double RPM7;
56      public double RPM8;
57
58      private void Awake()
59      {
60          //Asegura que se manda el dato de altitud del dron correcto, si no
                queda bloqueado a volar en vertical por la condicion suelo de SL
61          DroneSelected = PlayerPrefs.GetString("DroneSelected");
62          ActiveDron();
63          SoC = 100;
64
65          //Esto lanza el Modelo SL a traves de un fichero .bat localizado en
                la carpeta raiz del proyecto unity al iniciar el visor
66          Process.Start("Run");
67
68          _UdpReceiver = GetComponent<UDPReceiver>();
69          _UdpReceiver.SetObserver(this);
70          _Udpransmitter = GetComponent<UDPTransmitter>();
71
72
73      }
74
75      /// <summary>
76      /// Send data immediately after receiving it.
77      /// </summary>
78      /// <param name="val"></param>
79      void IReceiverObserver.OnDataReceived(double[] val)
80      {
81          //Aqui se leen los valores del enlace con Simulink, actualizando las
```

```
        variables del vector de estado,
82          //tambien aqui se aplican las conversiones necesarias para pasar de
        base SL a base Unity
83
84          // Posiciones: x unity = x SL
85          //             y unity = -z SL
86          //             z unity = y SL
87          // Angulos: yaw  = giro en y en unity = phi = -values [4]
88          //          pitch=  giro en z del unity = theta = values[5]
89          //          roll =  giro en x en unity = psi = values [6]
90
91          double[] values = _UdpReceiver.statevector;
92          simtime = values[0];
93          simX = values[1];
94          simY = -values[3];
95          simZ = values[2];
96          simyaw = -values[4] * Mathf.Rad2Deg;
97          simpitch = values[5] * Mathf.Rad2Deg;
98          simroll = values[6] * Mathf.Rad2Deg;
99          simspeed = new Vector3((float)values[7], (float)values[8], -(float)
        values[9]);
100         SoC = values[10];
101         RPM1 = values[11];
102         RPM2 = values[12];
103         RPM3 = values[13];
104         RPM4 = values[14];
105         RPM5 = values[15];
106         RPM6 = values[16];
107         RPM7 = values[17];
108         RPM8 = values[18];
109
110
111         //Aqui vas a leer los datos a enviar en el array correspondiente y
        los grabas como variables locales para construir el vector de
        subida
112         simAGL = Convert.ToDouble(ActiveDrone.heightAboveGround);
113         simWOW = Convert.ToDouble(ActiveDrone.groundCollision); //Weight On
        Wheels, deteccion de si toca el suelo
```

```
114         simSTOP = Convert.ToDouble(Botonera.simStop);
115         WindForce = Convert.ToDouble(Wind.WindForce);
116         WindDirection = Convert.ToDouble(Wind.WindDirection);
117         nfallos = FunctionsBotonera.nfallos;
118         RotorFallido = Convert.ToDouble(FunctionsBotonera.RotorFallido);
119         FunctionSelected = FunctionsBotonera.FunctionSelected;
120         HoverTarget = FunctionsBotonera.HoverTarget;
121         AutolandingMode = Convert.ToDouble(FunctionsBotonera.AutolandingMode)
                ;
122         //este es el array que se manda por UDP a Matlab
123         UpValues = new double[] {simAGL, simWOW, simSTOP, WindForce,
                WindDirection, nfallos, RotorFallido, FunctionSelected,
                HoverTarget, AutolandingMode};
124 
125         _Udpransmitter.Send(UpValues);
126 
127     }
128 
129     private void ActiveDron()
130     {
131         if (DroneSelected == "QUAD_WG")
132         {
133             ActiveDrone = quadcopter;
134         }
135         if (DroneSelected == "QUAD_WOG")
136         {
137             ActiveDrone = quadcopterWOG;
138         }
139         if (DroneSelected == "HEXA_")
140         {
141             ActiveDrone = hexacopter;
142         }
143         if (DroneSelected == "OCTO_")
144         {
145             ActiveDrone = octocopter;
146         }
147     }
148 }
```

A.2.3. CameraSwapper.cs

Este código prepara las cámaras virtuales Cinemachine en función del dron escogido y maneja el cambio de perspectivas como se ha descrito en el capítulo 5.

```
1
2 using System.Collections;
3 using System.Collections.Generic;
4 using UnityEngine;
5 using Cinemachine;
6
7 public class CameraSwapper : MonoBehaviour
8 {
9     public ButtonUI botonera;
10    public GameObject CamaraFija;
11    public GameObject CamaraSeguir;
12    public GameObject QuadWG;
13    public GameObject QuadWOG;
14    public GameObject Hexa;
15    public GameObject Octo;
16    public CinemachineVirtualCamera Fija;
17    public CinemachineVirtualCamera Seguir;
18    private string DroneSelected;
19
20    void Start()
21    {
22        DroneSelected = PlayerPrefs.GetString("DroneSelected");
23        DroneSetcamera();
24    }
25
26    void Update()
27    {
28        if (botonera.ActiveCam == true)
29        {
30            CamaraFija.SetActive(true);
31            CamaraSeguir.SetActive(false);
32        }
33        else if (botonera.ActiveCam == false)
34        {
```

```
35                CamaraSeguir.SetActive(true);
36                CamaraFija.SetActive(false);
37            }
38        }
39
40        void DroneSetcamera()
41        {
42            if (DroneSelected == "QUAD_WG")
43            {
44                QuadWG.SetActive(true);
45                QuadWOG.SetActive(false);
46                Hexa.SetActive(false);
47                Octo.SetActive(false);
48
49                Fija.LookAt = QuadWG.transform;
50                Seguir.LookAt = QuadWG.transform;
51                Seguir.Follow = QuadWG.transform;
52                Seguir.GetCinemachineComponent<CinemachineTransposer>().
                      m_FollowOffset = new Vector3(-1.0f, 0.2f, 0.21f);
53                Seguir.GetCinemachineComponent<CinemachineComposer>().
                      m_TrackedObjectOffset = new Vector3(0.0f, 0.0f, 0.21f);
54            }
55            if (DroneSelected == "QUAD_WOG")
56            {
57                QuadWG.SetActive(false);
58                QuadWOG.SetActive(true);
59                Hexa.SetActive(false);
60                Octo.SetActive(false);
61
62                Fija.LookAt = QuadWOG.transform;
63                Seguir.LookAt = QuadWOG.transform;
64                Seguir.Follow = QuadWOG.transform;
65                Seguir.GetCinemachineComponent<CinemachineTransposer>().
                      m_FollowOffset = new Vector3(-0.76f, 0.4f, 0.0f);
66                Seguir.GetCinemachineComponent<CinemachineComposer>().
                      m_TrackedObjectOffset = new Vector3(0.0f, 0.0f, 0.0f);
67            }
68            if (DroneSelected == "HEXA_")
```

```
69          {
70              QuadWG.SetActive(false);
71              QuadWOG.SetActive(false);
72              Hexa.SetActive(true);
73              Octo.SetActive(false);
74
75              Fija.LookAt = Hexa.transform;
76              Seguir.LookAt = Hexa.transform;
77              Seguir.Follow = Hexa.transform;
78              Seguir.GetCinemachineComponent<CinemachineTransposer>().
                    m_FollowOffset = new Vector3(-0.99f, 0.6f, 0.23f);
79              Seguir.GetCinemachineComponent<CinemachineComposer>().
                    m_TrackedObjectOffset = new Vector3(0.0f, 0.0f, 0.22f);
80          }
81          if (DroneSelected == "OCTO_")
82          {
83              QuadWG.SetActive(false);
84              QuadWOG.SetActive(false);
85              Hexa.SetActive(false);
86              Octo.SetActive(true);
87
88              Fija.LookAt = Octo.transform;
89              Seguir.LookAt = Octo.transform;
90              Seguir.Follow = Octo.transform;
91              Seguir.GetCinemachineComponent<CinemachineTransposer>().
                    m_FollowOffset = new Vector3(-2.21f, 1.11f, 0.21f);
92          }
93      }
94
95  }
```

A.2.4. Control del viento

Este código determina la fuerza y dirección del viento en función de lo seleccionado en el menú principal, en caso de haber seleccionado la opción de randomizar, se varía de manera aleatoria un en un 10 % los valores elegidos. Mas adelante, se incluyen los códigos correspondientes para la afeccción del viento sobre la manga de viento y para limitar la afección/representación del viento al entorno del dron.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class WindControl : MonoBehaviour
{
    private double WindDirection;
    private double WindForce;
    public WindZone Wind;
    public GameObject loopeffect;
    public GameObject waveeffect;
    private ParticleSystem loop;
    private ParticleSystem wave;

    private float randomforce;
    private float randomdir;
    private int WindDirRand;
    private int WindFRand;
    public float newWinddir;
    public float newWindF;

    // Start is called before the first frame update
    void Start()
    {
        WindDirection = 0;
        WindForce = 0;
        WindForce = (double)0.5f*1.225*Mathf.Pow(PlayerPrefs.GetFloat("
            WindForce"),2);
        WindDirection = (double)PlayerPrefs.GetFloat("WindDir");
        WindFRand = PlayerPrefs.GetInt("WindFRand");
        WindDirRand = PlayerPrefs.GetInt("WindDirRand");

        Wind.windMain = (float)WindForce;
        Wind.transform.rotation = Wind.transform.rotation * Quaternion.Euler(
            new Vector3(0f, (float)WindDirection, 0f));

        loop = loopeffect.GetComponent<ParticleSystem>();
```

```
36          wave = waveeffect.GetComponent<ParticleSystem>();
37 
38          loop.Play();
39          wave.Play();
40      }
41 
42      void Update()
43      {
44          RandomiseWind();
45          if (Wind.windMain > 0f)
46          {
47              loopeffect.SetActive(true);
48              waveeffect.SetActive(true);
49          }
50          if (Wind.windMain == 0f)
51          {
52              loopeffect.SetActive(false);
53              waveeffect.SetActive(false);
54          }
55      }
56 
57      private void RandomiseWind()
58      {
59          if (WindFRand == 1)
60          {
61          randomforce = Random.Range(-0.1f * (float)WindForce, 0.1f * (float)
               WindForce);
62          newWindF = (float)WindForce + randomforce;
63          Wind.windMain = newWindF;
64          }
65          if (WindDirRand == 1)
66          {
67              randomdir = Random.Range(-0.05f * (float)WindDirection, 0.05f * (
                   float)WindDirection);
68              newWinddir = (float)WindDirection + randomdir;
69              Quaternion oldWindDir = Wind.transform.rotation;
70              Wind.transform.rotation = Quaternion.RotateTowards(oldWindDir,
                   Quaternion.Euler(new Vector3(0f, newWinddir, 0f)), Time.
```

```
                deltaTime * 0.15f) ;
71          }
72          if (WindFRand == 0)
73          {
74              newWindF = (float)WindForce ;
75          }
76          if (WindDirRand == 0)
77          {
78              newWinddir = (float)WindDirection ;
79          }
80      }
81  }
```

Windsockrotate.cs

Este código programa la dirección en la que debe estar la manga de viento y la fuerza con la que se mueve en función de lo seleccionado el menú principal. A fin de darle cierto realismo, se añade una fuerza aleatoria (no modelada en la simulación del dron) que hace que se mueva con cierta turbulencia. Asimismo, durante la simulación la posición y rotación se actualizan en base a la evolución del viento.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class windsockrotate : MonoBehaviour
6  {
7          public Cloth manga;
8      public GameObject WindSock;
9      public WindControl wind;
10     private float WindDirection;
11     private float WindForce;
12
13     void Start()
14     {
15         WindDirection = PlayerPrefs.GetFloat("WindDir");
16         WindForce = 0.5f * 1.225f * Mathf.Pow(PlayerPrefs.GetFloat("WindForce
               "), 2);
17         WindSock.transform.rotation = WindSock.transform.rotation *
```

```
            Quaternion.Euler(new Vector3(0f, WindDirection, 0f));
18
19          manga.externalAcceleration = new Vector3(WindForce, 0f, 0f);
20          manga.randomAcceleration = new Vector3(0.1f * WindForce, 0.1f *
                WindForce, 0.1f * WindForce);
21      }
22
23      void Update()
24      {
25          WindDirection = wind.newWinddir;
26          WindForce = wind.newWindF;
27          Quaternion oldDir = WindSock.transform.rotation;
28          WindSock.transform.rotation = Quaternion.RotateTowards(oldDir,
                Quaternion.Euler(new Vector3(0f, WindDirection, 0f)), Time.
                deltaTime * 0.15f);
29
30          manga.externalAcceleration = new Vector3(WindForce, 0f, 0f);
31          manga.randomAcceleration = new Vector3(0.1f * WindForce, 0.1f *
                WindForce, 0.1f * WindForce);
32      }
33  }
34  }
```

A.2.5. Giro de las hélices y sonido de motores

Estos dos código actualizan en tiempo real, en base a la información recibida por el componente de comunicaciones, la velocidad angular de las hélices y modulan el sonido de los motores en base a éste. Aquí se presenta el código genérico, pues este es particularizado para los tres casos: quad, hexa y octo como parte del proceso de configuración/implementación del dron en Unity.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class BladeSpin_N : MonoBehaviour
6  {
7      public Rigidbody pala1;
8      public Rigidbody pala2;
```

```
9       public Rigidbody pala(...);
10      public Rigidbody palaN;
11
12      private Vector3 AngVel;
13      public CommunicationController Comms;
14
15      void Start()
16      {
17          AngVel = new Vector3(0f, 0f, 0f);
18          pala1.maxAngularVelocity = 1000;
19          pala2.maxAngularVelocity = 1000;
20          pala(...).maxAngularVelocity = 1000;
21          palaN.maxAngularVelocity = 1000;
22      }
23
24      //Este metodo sirve para simplificar al ser N palas, simplemente copiar y
            llamar para cada pala.
25      private void SpinThat(float Spin, Rigidbody pala)
26      {
27          AngVel = new Vector3(0f, Spin, 0f); //se recuerda que el eje y es el
                vertical en Unity
28          pala.angularVelocity = AngVel * 1.0f;
29      }
30
31      private void ActualizarAngVel()
32      {
33          float Spin1 = (float)Comms.RPM1;
34          float Spin2 = (float)Comms.RPM2;
35          float Spin(...) = (float)Comms.RPM(...);
36          float SpinN = (float)Comms.RPMN;
37
38          SpinThat(Spin1, pala1);
39          SpinThat(Spin2, pala2);
40          SpinThat(Spin3, pala(...));
41          SpinThat(Spin4, palaN);
42      }
43
44      void Update()
```

```
45      {
46          ActualizarAngVel();
47      }
48  }
```

Control de audio de las hélices

Este sencillo código modula el tono del sonido de motores en función de la velocidad angular de la hélice.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class audiocontrol_prop : MonoBehaviour
6  {
7      AudioSource audioSource;
8      public Rigidbody prop;
9
10     void Start()
11     {
12         audioSource = GetComponent<AudioSource>();
13         audioSource.pitch = 1;
14     }
15
16     void Update()
17     {
18         audioSource.pitch = prop.angularVelocity.y / 600;
19     }
20 }
```

A.2.6. Interfaz gráfica

En esta sección se muestran los códigos que rigen los distintos elementos descritos en 5.

IU: Datos de vuelo

Este código controla la posición de las manecillas de los relojes, así como el color y tamaño del indicador de batería.

```
1  using System.Collections;
```

```
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.UI;
5
6 public class flightdataUI : MonoBehaviour
7 {
8     public RectTransform speedneedle;
9     private Vector3 simspeed;
10    private float Velocidad;
11    private float RoC;
12    private float AnguloVelocidad;
13
14
15    public RectTransform AGLneedle;
16    private float AGL;
17    private float AnguloAGL;
18
19    public Scrollbar BatteryBar;
20    public float SoC;
21    private ColorBlock batterycolor;
22
23    public RectTransform RPMneedle;
24    public float RPMsum;
25    public float RPMavg;
26    public float AnguloRPM;
27
28    public CommunicationController Comms;
29    private string DroneSelected;
30    private int nrotors;
31
32
33    void Start()
34    {
35        speedneedle.SetAsLastSibling(); //Esto permite que la aguja este
              siempre por encima del resto del reloj
36        AGLneedle.SetAsLastSibling();
37        RPMneedle.SetAsLastSibling();
38
```

```
39          Rotores();
40
41          if (nrotors == 4)
42          {
43              GameObject.Find("BatteryLeft").SetActive(false); //En el Quad no
                    se ha modelado la bateria
44          }
45
46          SoC = 1;
47          batterycolor = BatteryBar.colors;
48          batterycolor.disabledColor = Color.green;
49      }
50
51      // Update is called once per frame
52      void Update()
53      {
54          UpdateVel();
55          UpdateAGL();
56          UpdateSoC();
57          UpdateRPM();
58      }
59
60      private void UpdateVel()
61      {
62          simspeed = Comms.simspeed;
63          Velocidad = simspeed.magnitude;
64          AnguloVelocidad = (Velocidad / 10f) * 225f; //el 10 es el fondo de
                escala del reloj
65
66          speedneedle.rotation = Quaternion.Euler(0f, 0f, -Mathf.Clamp(
                AnguloVelocidad, 0f, 225f));
67
68      }
69
70      private void UpdateAGL()
71      {
72          AGL = (float)Comms.simAGL;
73          AnguloAGL = (AGL / 150f) * 225f; //aqui se define el fondo de escala
```

```
        del reloj, en nuestro caso estamos hablando de drones que deben
        volar por debajo de 120m segun normativa por lo que este valor es
        aceptable.
74          AGLneedle.rotation = Quaternion.Euler(0f, 0f, -Mathf.Clamp(AnguloAGL,
                0f, 225f));
75      }
76
77      private void UpdateSoC()
78      {
79          SoC = (float)Comms.SoC;
80          BatteryBar.size = SoC/100;
81          BatteryBar.value = 0.01f;
82          BatteryBar.value = 0f;
83          if (SoC < 15)
84          {
85          Debug.Log("<color=red>" + "Low Battery!" + "</color>");
86          }
87      }
88
89      public void OnSoCchange()
90      {
91          Color newcolor = Color.Lerp(Color.red, Color.green, SoC/100);
92          batterycolor.disabledColor = newcolor;
93          BatteryBar.colors = batterycolor;
94      }
95
96      private void UpdateRPM()
97      {
98         float absrpm1 = Mathf.Abs((float)Comms.RPM1) * 9.5492968f;
99         float absrpm2 = Mathf.Abs((float)Comms.RPM2) * 9.5492968f;
100        float absrpm3 = Mathf.Abs((float)Comms.RPM3) * 9.5492968f;
101        float absrpm4 = Mathf.Abs((float)Comms.RPM4) * 9.5492968f;
102        float absrpm5 = Mathf.Abs((float)Comms.RPM5) * 9.5492968f;
103        float absrpm6 = Mathf.Abs((float)Comms.RPM6) * 9.5492968f;
104        float absrpm7 = Mathf.Abs((float)Comms.RPM7) * 9.5492968f;
105        float absrpm8 = Mathf.Abs((float)Comms.RPM8) * 9.5492968f;
106
107         RPMsum = absrpm1 + absrpm2 + absrpm3 + absrpm4 + absrpm5 + absrpm6 +
```

```
                absrpm7 + absrpm8;
108         RPMavg = RPMsum / nrotors;
109         AnguloRPM = (RPMavg / 9000f) * 225f;
110         RPMneedle.rotation = Quaternion.Euler(0f, 0f, -Mathf.Clamp(AnguloRPM,
                 0f, 225f));
111     }
112
113     private void Rotores()
114     {
115         DroneSelected = PlayerPrefs.GetString("DroneSelected");
116         if (DroneSelected == "QUAD_WG")
117         {
118             nrotors = 4;
119         }
120         if (DroneSelected == "QUAD_WOG")
121         {
122             nrotors = 4;
123         }
124         if (DroneSelected == "HEXA_")
125         {
126             nrotors = 6;
127         }
128         if (DroneSelected == "OCTO_")
129         {
130             nrotors = 8;
131         }
132     }
133 }
```

IU: Datos de entrada de mando

Este código controla la posición de los sticks en el mando de la interfaz, así como el indicador de mando conectado.

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.UI;
5
```

```
public class UIcontrollerPos : MonoBehaviour
{
    private float indicadorsize;
    public RectTransform UIstick1;
    public RectTransform UIstick2;
    public string[] joysticks;
    public RawImage connectionImg;

    void Update()
    {
        joysticks = Input.GetJoystickNames();

            if (joysticks[0] != null)
        {
            connectionImg.color = Color.green;
            UIstick1.anchoredPosition = new Vector2(40.0f * Input.GetAxis("
               Horizontal"), 40.0f * Input.GetAxis("Vertical"));
            UIstick2.anchoredPosition = new Vector2(40.0f * Input.GetAxis("
               Horizontal2"), 40.0f * Input.GetAxis("Vertical2"));
        }
            if (joysticks[0] == "")
        {
            connectionImg.color = Color.red;
            UIstick1.anchoredPosition = new Vector2(0.0f, 0.0f);
            UIstick2.anchoredPosition = new Vector2(0.0f, 0.0f);
        }
    }
}
```

IU: Brújula

Este código controla la la transformación aplicada al gráfico que sirve como brújula para indicar la dirección del dron, asumiendo el norte como orientación inicial.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

```

```
6  public class UICompass : MonoBehaviour
7  {
8      public RawImage compassImage;
9      public Transform dron;
10
11     private void Update()
12     {
13         compassImage.uvRect = new Rect(dron.localEulerAngles.y / 360f, 0f, 1f
               , 1f);
14     }
15 }
```

IU: Datos de simulación

Este código controla los textos mostrados en el grupo de datos de la simulación de la interfaz.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.UI;
5  using TMPro;
6
7  public class TextUI : MonoBehaviour
8  {
9
10     public TextMeshProUGUI TiempoText;
11     public TextMeshProUGUI PosText;
12     public CommunicationController Comms;
13     private string timestring;
14     private double time;
15     private float timemin;
16     private float timesec;
17     private string posstring;
18     private Vector3 pos3d;
19
20     static string myLog = "";
21     private string output;
22     private string stack;
23     public TextMeshProUGUI LogText;
```

```
24
25      void Update()
26      {
27          time = Comms.simtime;
28          timemin = Mathf.Floor((float)time /60);
29          timesec = Mathf.Floor((float)time - timemin*60);
30          timestring = timemin.ToString() + ":" +timesec.ToString();
31          TiempoText.text = timestring;
32
33          pos3d = new Vector3((float)Comms.simX, (float)Comms.simY, (float)
                Comms.simZ);
34          posstring = pos3d.ToString();
35          PosText.text = posstring;
36
37          LogText.text = myLog;
38      }
39
40  //Esta parte del codigo permite leer el Log predeterminado de Unity y lo
        graba en la variable myLog
41
42      void OnEnable()
43      {
44          Application.logMessageReceived += Log;
45      }
46
47      void OnDisable()
48      {
49          Application.logMessageReceived -= Log;
50      }
51
52      public void Log(string logString, string stackTrace, LogType type)
53      {
54          output = logString;
55          stack = stackTrace;
56          myLog = output + "\n" ;
57      }
58  }
```

IU: Botonera Principal

Este código controla las acciones de algunos de los botones en la botonera principal, cabe destacar que otros estan definidos de manera directa sobre el propio botón (sin código) utilizando las herramientas visuales de Unity.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class ButtonUI : MonoBehaviour
{
    public Button StopButton;
    public Button MenuButton;
    public Button CamButton;
    public GameObject FunctionsMenu;
    public bool simStop;
    public bool MenuBool;
    public bool ActiveCam;
    public CommunicationController Comms;
    void Start()
    {
        // Aqui se inicializan los botones para detectar click
        simStop = false;
        ActiveCam = false;
        MenuBool = false;
        FunctionsMenu.transform.localScale = new Vector3(0, 0, 0);
        StopButton.onClick.AddListener(TaskOnClick0);
        CamButton.onClick.AddListener(TaskOnClick2);

    }

    // Cuando clickes en el boton se hace:
    void TaskOnClickStop()
    {
        Debug.Log("You have clicked the stop button!"); //esto avisara al
            usuario de que en efecto se ha parado la simulacion
```

```
33        if (simStop == false)
34        {
35            simStop = true;
36        }
37        else
38        {
39            simStop = false;
40        }
41    }
42    public void BackToMenuButton()
43    {
44        simStop = true;
45        Comms.simSTOP = 1;
46        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex - 1);
47    }
48
49    void TaskOnClickSwap()
50    {
51        if (ActiveCam == true)
52        {
53            ActiveCam = false;
54        }
55        else if (ActiveCam == false)
56        {
57            ActiveCam = true;
58        }
59    }
60
61    public void StartSim()
62    {
63        Debug.Log("Launching Simulation, please wait!");
64        simStop = false;
65        System.Diagnostics.Process.Start("Run");
66    }
67
68    public void ToggleFunctionsMenu()
69    {
70        if(MenuBool == true)
```

```
71          {
72              FunctionsMenu.transform.localScale = new Vector3(0, 0, 0);
73              //se decide ocultarlo bajando su escala en lugar de desactivar el
                    gameobject ya que, de desactivarlo, se inutilizarian los
                    scripts asociados
74              MenuBool = false;
75          }
76          else if (MenuBool == false)
77          {
78              FunctionsMenu.transform.localScale = new Vector3(1, 1, 1);
79              MenuBool = true;
80          }
81      }
82
83  }
```

IU: Funciones adicionales

Este código controla las acciones de algunos de los botones en la botonera desplegable, así como los campos para introducir el Hover Target y el desplegable para seleccionar la dirección de vuelo asistido.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.UI;
5   using TMPro;
6
7   public class FunctionsButtonUI : MonoBehaviour
8   {
9       private string DroneSelected;
10      public double FunctionSelected;
11      public bool AutolandingMode;
12      public int RotorFallido;
13      public double nfallos;
14      private int nrotors;
15      public double HoverTarget;
16      public GameObject HoverTgtInput;
17      private Dropdown FAModeDropdown;
```

```
18      public CommunicationController Comms;
19
20
21      void Start()
22      {
23          DroneSelected = PlayerPrefs.GetString("DroneSelected");
24          Rotores();
25          RotorFallido = (int)Random.Range(1, nrotors + 1);
26          FAModeDropdown = GameObject.Find("FAModes").GetComponent<Dropdown>();
27          AutolandingMode = false;
28      }
29
30      void Update()
31      {
32          if(Comms.SoC < 15 && PlayerPrefs.GetInt("FTS") == 1)
33          {
34              AutolandingMode = true;
35              //esta condicion supone el FTS incorporado. Por lo general, las
                    baterias utilizadas para drones no deben descargarse
                    completamente ya que pierden capacidad, por lo que se decide
                    activar el autoaterrizaje a partir del 15%. Asimismo, se
                    incorpora una medida para desactivar este sistema en el menu
                    de opciones por si fuera necesario.
36          }
37      }
38
39      public void OnClickManual()
40      {
41          FunctionSelected = 0;
42          FAModeDropdown.value = 0;
43      }
44
45      public void OnClickHover()
46      {
47          FunctionSelected = 5;
48          FAModeDropdown.value = 5;
49      }
50
```

```
51      public void OnClickAutolanding()
52      {
53          if (AutolandingMode == true)
54          {
55              AutolandingMode = false;
56          }
57          else if (AutolandingMode == false)
58          {
59              AutolandingMode = true;
60          }
61
62      }
63
64      public void OnClickFallo()
65      {
66          if (nfallos<nrotors)
67          {
68          nfallos = nfallos + 1;
69          }
70          Debug.Log("You have clicked the Fallo button, nfallos:" + nfallos.
                ToString());
71      }
72
73      public void OnClickResetFallo()
74      {
75          nfallos = 0;
76          RotorFallido = (int)Random.Range(1, nrotors + 1);
77      }
78
79      private void Rotores()
80      {
81          if (DroneSelected == "QUAD_WG")
82          {
83              nrotors = 4;
84          }
85          if (DroneSelected == "QUAD_WOG")
86          {
87              nrotors = 4;
```

```
88          }
89          if (DroneSelected == "HEXA_")
90          {
91              nrotors = 6;
92          }
93          if (DroneSelected == "OCTO_")
94          {
95              nrotors = 8;
96          }
97      }
98
99      public void OnHoverTargetChange()
100     {
101         HoverTarget = System.Convert.ToDouble(HoverTgtInput.GetComponent<
                TMP_InputField>().text);
102     }
103
104     public void OnFAModeChange()
105     {
106         FunctionSelected = FAModeDropdown.value;
107     }
108 }
```