Trabajo Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica
Mención en Automática y Robótica

Modelado y control de quadrotors en la plataforma UNITY 3D

Autor: Javier Gómez Jiménez
Tutor: Juan Manuel Escaño González
Cotutor: Adolfo J. Sánchez del Pozo Fernández

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica
Mención en Automática y Robótica

Modelado y control de quadrotors en la plataforma UNITY 3D

Autor:
Javier Gómez Jiménez

Tutor:
Juan Manuel Escaño González
Cotutor:
Adolfo J. Sánchez del Pozo Fernández

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo Fin de Grado: Modelado y control de quadrotors en la plataforma UNITY 3D

Autor: Javier Gómez Jiménez

Tutor: Juan Manuel Escaño González

Cotutor: Adolfo J. Sánchez del Pozo Fernández

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A todos aquellos que nunca
dejaron de creer en mí

AGRADECIMIENTOS

A mis Amigos, porque en estos cuatro años no ha habido ningún momento que haya estado solo, pasara lo que pasase. Siempre habéis sabido acompañarme, darne ánimos, ayudarme tanto en lo técnico como en lo humano, que ahora es lo que más valoro.

A mi Familia, por quererme tal como soy, y no por mis actos.

A mi profesor Manuel Gil Ortega Linares, por meter en mi vida el mundo del Control Automático.

A mi cotutor Adolfo J., por hacerme pasar unos ratos inolvidables, y aportarme todo el conocimiento técnico que necesitaba para afrontar este proyecto.

A mi tutor Juan Manuel Escaño, por hacer posible mis ideas, por muy difíciles que parecieran. Y, sobre todo, por estar ahí cuando no había luz.

Y a mi Padre, por conseguir en minutos lo que no he conseguido en semanas.

Javier Gómez Jiménez

Alumno de último curso de Ingeniería Electrónica, Robótica y Mecatrónica

Sevilla, 2020

RESUMEN

En el presente proyecto se ha diseñado y programado un simulador escalable virtual para las pruebas de quadrotors, o coloquialmente conocidos como "drones", para conseguir un entorno fiable, eliminando costes adicionales que conllevan las pruebas vuelos, así como los asuntos relacionados con la seguridad.

Se ha diseñado la arquitectura de control y el generador de trayectorias. Se ha diseñado un modelo en 3D y creado el entorno de simulación en el software Unity3d.

ABSTRACT

In this project, a virtual scalable simulator for quadrotor tests, or colloquially known as "drones", has been designed and programmed to achieve a reliable environment, eliminating additional costs associated with flight tests, as well as security-related issues .

The control architecture and the path generator have been designed. A 3D model has been designed and the simulation environment created in the Unity3d software.

ÍNDICE

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

En estos últimos años han ocurrido crecimientos exponenciales en muchos campos de la ingeniería. Entres ellos se pueden nombrar los campos de la electrónica, la robótica, del control automático, de la informática, de la mecánica, etc. Poco a poco se han ido rompiendo las barreras que el hombre no se había ni imaginado traspasar.

Todo esto implica un esfuerzo por desarrollar todas las materias que acompañan a estas disciplinas. Poniendo un ejemplo: si teóricamente se desarrolla la arquitectura de un algoritmo de procesamiento de imágenes que utiliza inteligencia artificial y *machine learning*, pero por el contrario el campo del hardware no avanzase, no se podría implementar en la práctica.

Este proyecto sirve de apoyo a todos los avances que ha habido en la robótica aérea; muy en concreto en el campo de la simulación y pruebas que ésta necesita. A su vez, este proyecto no hubiera sido posible sin los grandes recorridos que ya existen en el mundo de la informática, con simuladores físicos virtuales que permiten llevar parte de la física real dentro de un ordenador.

Entornos físicos existen muchos, como pueden ser Unreal Engine, Blender, Unity3d, etc. En general, las diferencias son múltiples, como pueden ser los motores físicos, integración de gráficos, física de choques… Estos entornos se utilizan mayoritariamente para realizar videojuegos, pero tienen otras utilidades dentro del mercado. Un ejemplo es en el marketing de empresas, ofreciendo visualizar a sus clientes los productos con unas gafas de realidad virtual. Otro ejemplo de utilidad es el presente proyecto: utilizar los motores físicos de estos entornos para simular os comportamientos de robots aéreos.

Existen múltiples proyectos que introducen vehículos aéreos dentro de estos motores. La gran diferencia entre ellos es la finalidad: algunos desean introducir un dron, otros simular de una manera más real el movimiento de este, otros incluso introducen controladores para controlar a bajo nivel el dron, etc. Como se puede ver, hay un grado de realidad dependiendo de la finalidad que se busque.

Algunos ejemplos de estos proyectos pueden ser el entorno de simulación creado por Mario Haberle [1]. En la figura 1.1 se puede observar un fotograma de su simulador. Éste consiste en modelar la cinemática de un quadrotor, pero sin incluir ningún control ni una dinámica. Los cambios en el quadrotor se realizan cambiando su posición y sus ángulos directamente, en ausencias de fuerzas. El resultado es un quadrotor que se puede controlar mediante entradas por teclado. Se podría decir que es el nivel más bajo de realidad en un modelo. Pero cumple su objetivo perfectamente, dado que su funcionalidad es simular un vehículo que vuele y eso es lo que consigue.

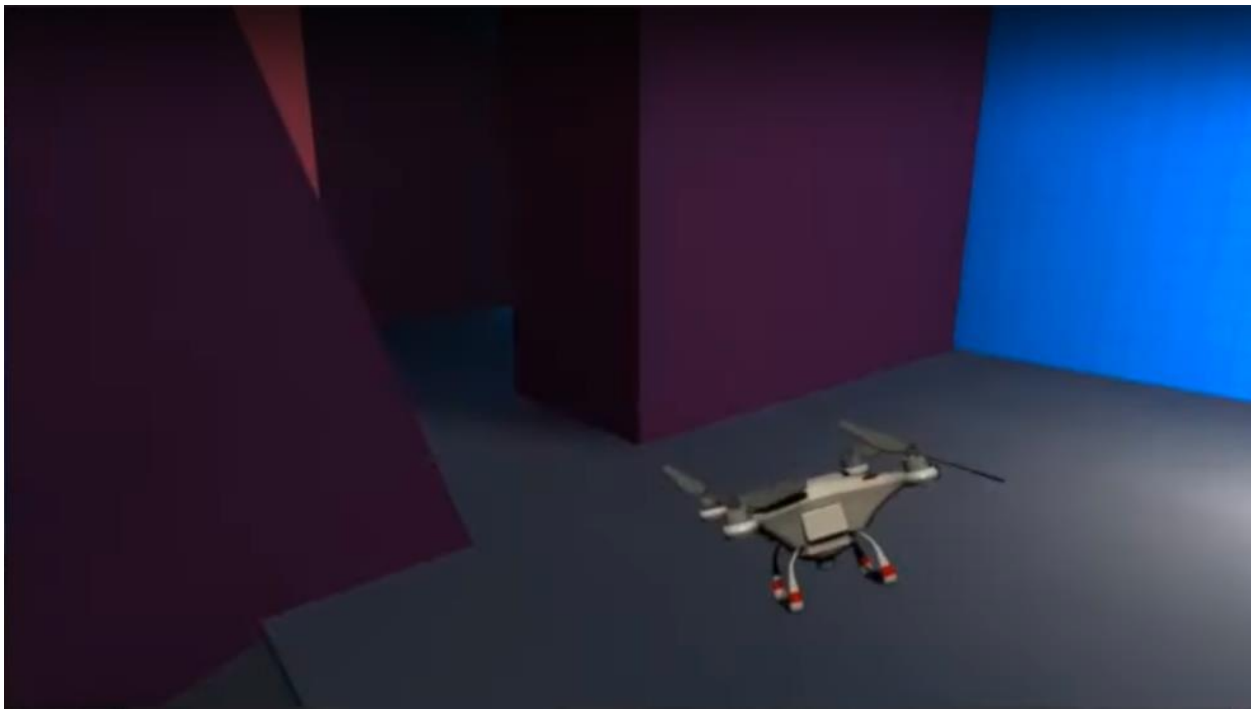

Figura 1.1 - Entorno de simulación de Mario Haberle

Otro ejemplo , subiendo el nivel de realismo, es el proyecto realizado por Erik Nordeus (Ejemplo en la figura 1.2) [2]. Erik introduce ya una dinámica, es decir, actúa sobre el quadrotor mediante la adición de fuerzas y pares, y no cambiando la posición ni los ángulos. Otro punto realista es la aplicación de las fuerzas, separando las fuerzas y aplicándolas en cada propulsor. Además, introduce un controlador a nivel bajo para controlar los ángulos y la altura. Al igual que el anterior, el quadrotor se controla presionando las teclas de dirección del teclado.

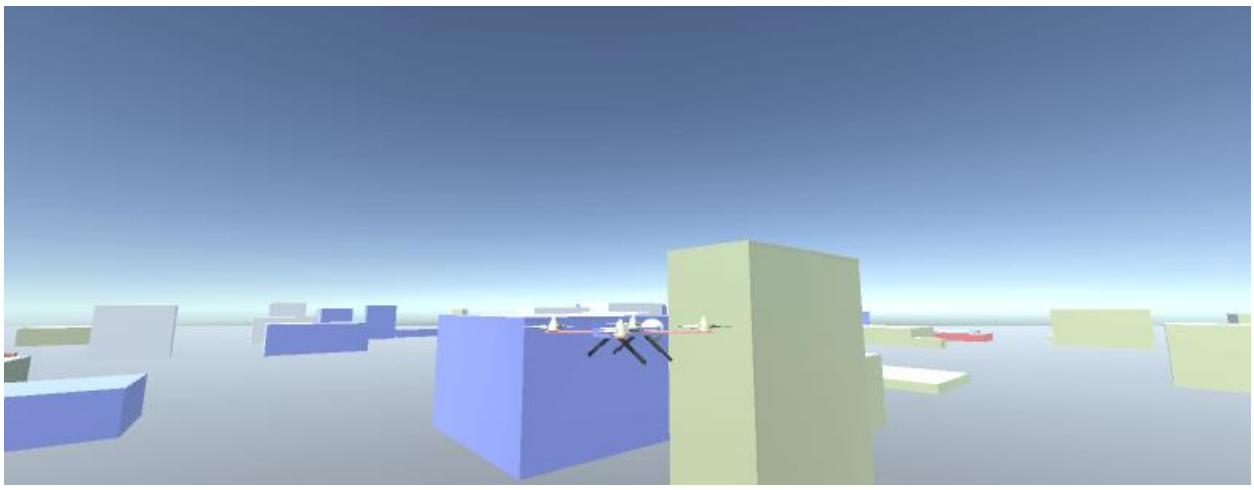

Figura 1.2 - Entorno de simulación de Erik Nordeus

Estos proyectos tienen limitaciones de realismo, pero es un camino docente práctico para ir aumentando el acercamiento a la realidad, hasta conseguir un *Gemelo Digital*, es decir, llegar a tener un simulador con el que predecir los comportamientos de un sistema real, siendo en el caso que nos compete un quadrotor [3].

La motivación de este proyecto es dar el siguiente paso en este camino, creando un simulador que permita probar y predecir los comportamientos de un quadrotor. A modo de recordatorio, las pruebas de un quadrotor tienen riesgos que a veces no se pueden asumir. Más arriba se habla de riesgos económicos y de seguridad. Para visualizar estos riesgos se expone a continuación un ejemplo: un quadrotor con funciones dentro de una fábrica donde trabajan humanos a la vez que el quadrotor trabaja, no permite hacer pruebas realistas sin sufrir graves consecuencias, como podría ser dañar la salud de cualquier trabajador, estropear el funcionamiento de alguna máquina o dejar inutilizable el quadrotor. Estos riesgos justifican la motivación de este proyecto, dado que, en gran medida, busca solucionarlos, entre otros problemas.

Una vez visto la motivación del proyecto, los dos proyectos anteriores, y la diferencia de utilización del motor físico en cada uno de los proyectos (prácticamente nula en el primer caso, y un nivel alto en el segundo caso) es más fácil ver las implicaciones que conllevan los objetivos del presente proyecto.

1.1 Objetivos del Proyecto

El presente proyecto persigue de forma general los siguientes objetivos:

- Crear un entorno de simulación de quadrotors, introduciendo todas las limitaciones que existen en la realidad.
- Crear un modelo dinámicamente y cinemáticamente realistas de cada quadrotor.
- Crear un control de varios niveles para controlar el quadrotor, con las restricciones reales.
- Cumplir los dos objetivos anteriores permitiendo la máxima escalabilidad y ampliabilidad del simulador.

Este proyecto no busca realizar, entre otras cosas, el mejor controlador de un quadrotor, pero sí que permita implementarse de manera fácil dentro del simulador.

Dado que se trabaja sobre un entorno virtual, todas las pruebas están exentas de cualquier riesgo, salvo el de la no similitud hacia la realidad. Por eso, éste es el primer objetivo.

1.2 Desarrollo del presente documento

Este proyecto se ha desarrollado cronológicamente de esta manera:

1. En primer lugar, partiendo de un modelo real, se ha diseñado la arquitectura de control en el programa Matlab. También se ha diseñado el generador de trayectorias en este software.
2. En Segundo lugar, una vez probado el buen funcionamiento de todos los elementos que forman el software de un quadrotor, se ha diseñado un modelo en 3d en el programa Cinema 4D.
3. Por ultimo se ha creado el entorno de simulación en el software Unity3d.

El presente documento redactará el contenido del proyecto siguiendo la anterior cronología.

2 MODELADO Y CONTROL DEL QUADROTOR

Para el desarrollo del modelado y el control del quadrotor se ha usado en este trabajo la herramienta Matlab. Mirando la propia página web, Matlab s*e utiliza para aprendizaje automático, procesamiento de señales, procesamiento de imágenes, visión artificial, comunicaciones, finanzas computacionales, diseño de control, robótica y muchos otros campos* [4].

Resumiendo, y concretando, Matlab se ha utilizado en este proyecto como herramienta para probar y visualizar todos los controladores y los diferentes planificadores aplicados a un modelo. Matlab incluye un entorno de simulación, llamado Simulink. En este entorno se ha creado un simulador para verificar todo.

Se ha utilizado la versión R2017a.

A modo de esquema, en Matlab se han realizado las siguientes tareas:

- Creación de un simulador en Simulink.
- Inserción de un modelo con un controlador de bajo nivel.
- Control de inversión.
- Controlador de estabilidad.
- Controlador de posición.
- Generadores de trayectorias: lineal y Smart.
- Diseño de los controladores.
- Simulación.

2.1 Creación de un simulador en la herramienta Simulink

El simulador refleja la arquitectura de control diseñada para controlar el quadrotor. Existen infinidad de técnicas de control. Algunas buscan linealizar el modelo, para aplicar técnicas de control clásicas a sistemas no lineales, como es un quadrotor. Algunas de estas técnicas son control por compensación de la gravedad, por compensación de la dinámica o por par calculado. Se han estudiado estas posibilidades y se hablará más adelante de la posibilidad de implementación en este proyecto. Pero en este caso se ha decidido utilizar una arquitectura de control en cascada, controlando en varios niveles todos las variables del quadrotor.

La arquitectura en cascada lleva consigo una restricción importante, a tener en cuenta en el diseño de los controladores (se hablará más adelante): el tiempo de respuesta de un controlador en un nivel debe ser más lento que cualquier controlador de más bajo nivel. Esto tiene sus excepciones, pero por ahora es suficiente.

En el simulador podemos encontrar los siguientes controladores, ordenados de menor nivel a mayor:

1. Control de bajo nivel: consigue adecuar las velocidades de cada motor a las velocidades de referencia. En el simulador diseñado en Matlab, está contenido en el modelo del Quad.
2. Control de inversion: no es propiamente un controlador. Consigue convertir los pares virtuales de referencia en los tres ángulos y en la altura a la velocidad de referencia para cada motor.

3. Control de estabilidad: controla los pares virtuales en los tres ángulos más la altura con respecto a las referencias para los tres ángulos más la altura.
4. Control de posición: implementa el control en el plano XY del espacio tridimensional. Sus entradas son las referencias de X e Y; y sus salidas son las referencias a los ángulos Roll y Pitch (se explicará más adelante a qué nos referimos con estos ángulos).
5. Planificador: o generador de trayectorias, dado que en el simulador de Simulink solo se ha buscado verificar el buen funcionamiento de los controladores. Directamente no es un controlador, pero más adelante cuando se vea el diseño del generador de trayectorias *inteligente* se verá que éste implementa un control en velocidad y aceleración.

De forma gráfica se pueden observer en la figura 2.1 estos 5 niveles.

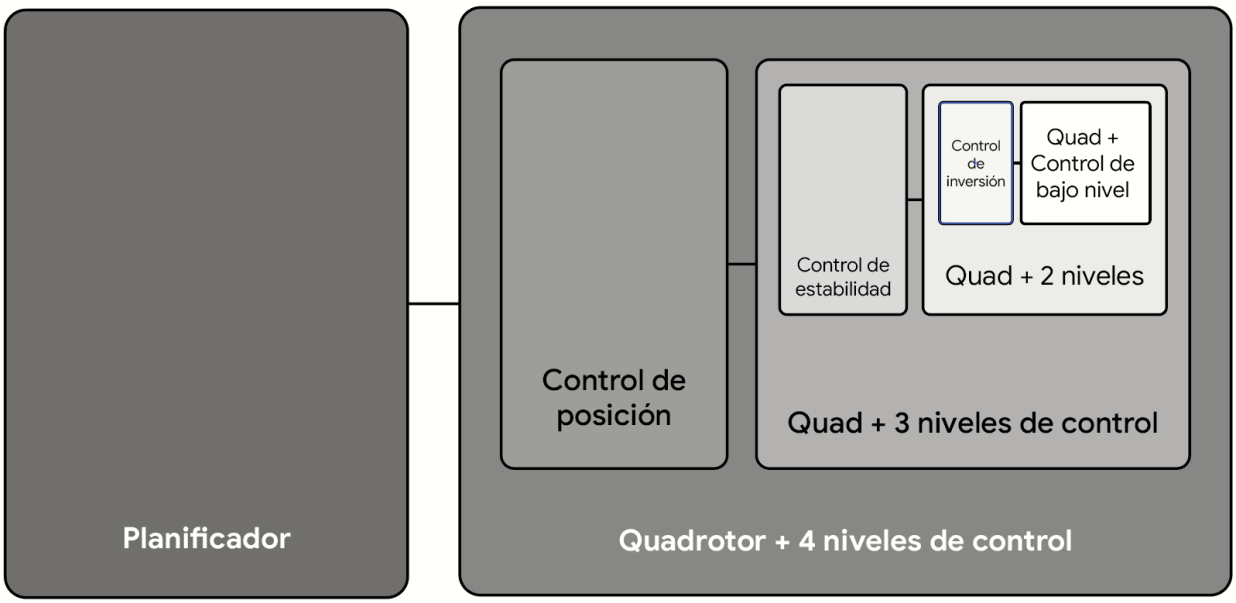

Figura 2.1 - Arquitectura de control implementada

A continuación, se expondrán imágenes reales del simulador en Simulink para analizar todos los elementos incluidos en el simulador. Se presentarán de mayor a menor nivel de profundidad.

En primer lugar, en la Figura 2.2 podemos ver una vista general del simulador, donde se puede observar el generador de trayectoria, donde la única entrada es el tiempo; y el dron con los cuatro niveles de control mencionados anteriormente. Se puede observar los 4 grados de libertad con los que podemos definir la posición de un quadrotor (entradas del QUAD). Restan dos ángulos, que debido a que el sistema es inestable, no podemos mantener una posición estable si esos dos ángulos (roll y pitch) no son nulos.

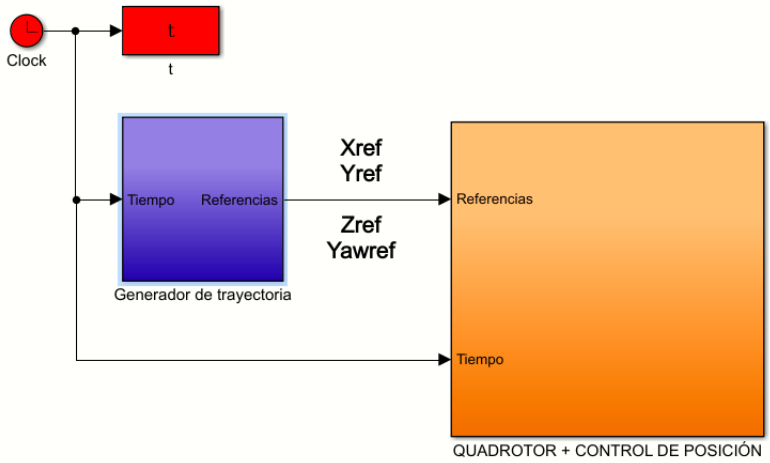

Figura 2.2 - Vista general del simulador en Simulink

Si entramos dentro del simulador (mírese la figura 2.3) podemos observar que tenemos varios parámetros y, en serie, el generador de splits para cada instante de la trayectoria. El generador de trayectorias es puntual, es decir, recibe un punto objetivo y no genera ningún punto intermedio de referencia, sino que busca unir, mediante los splits, el punto inicial y el final. Con esto es suficiente para poder probar los controladores que hay por debajo.

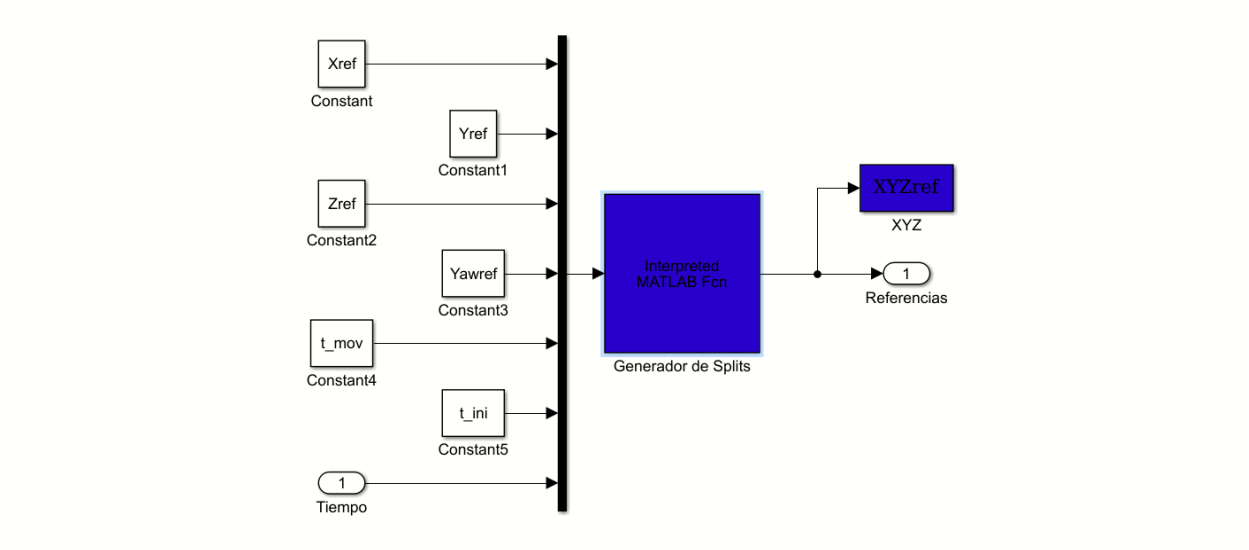

Figura 2.3 - Generador de trayectoria por dentro

Los parámetros que nos encontramos en el generador son los siguientes:

- Xref, Yref, Zref y Yawref: definen el punto final o punto objetivo.
- T_mov: contiene en segundos la duración del movimiento.
- T_ini: contiene en segundos cuando empezará el movimiento.

El generador de splits genera una línea recta de puntos intermedios entre el punto inicial y final. Estos puntos intermedios son los que devuelve.

Si entramos en el bloque QUADROTOR + CONTROL DE POSICIÓN, obsérvese la figura 2.4, nos encontramos por un lado el control de posición y el quadrotor con el control de estabilidad como control de más nivel.

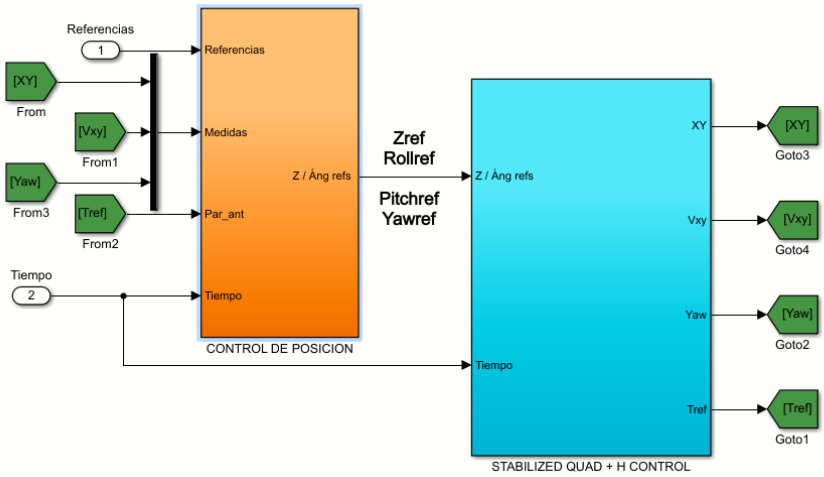

Figura 2.4 - Detalle del quadrotor más el control de posición

En este caso, se puede observar sin entrar en el bloque del controlador, las entradas que recibe éste.

Entrando aún más en el quadrotor (bloque STABILIZED QUAD + H CONTROL) nos encontramos lo que

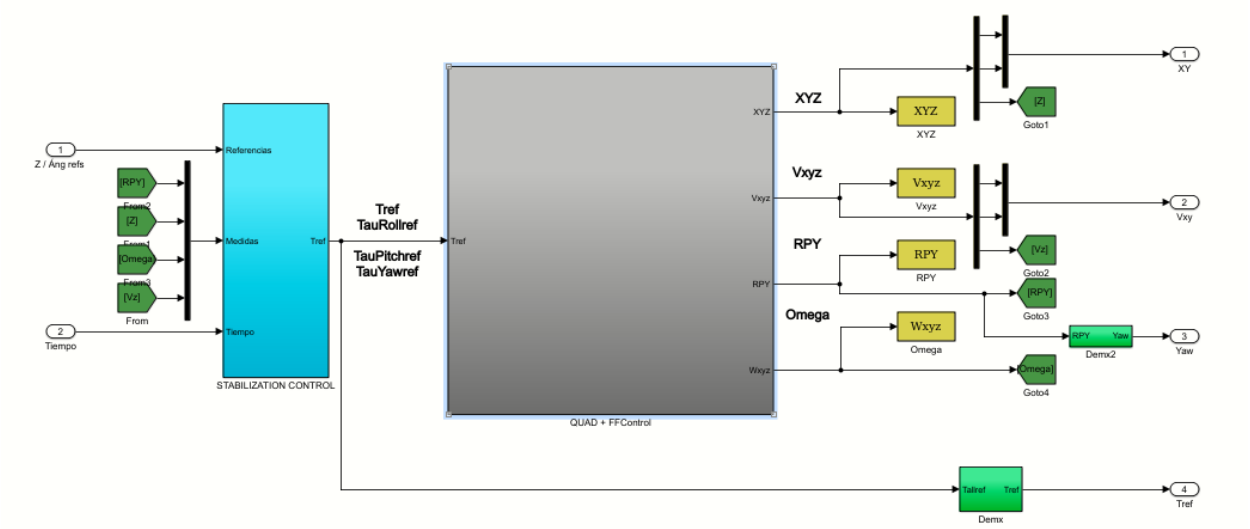

Figura 2.5 - Detalle del quadrotor más el control de estabilización

aparece en la figura 2.5.

Al igual que en el anterior caso, también se pueden apreciar las entradas del controlador de estabilidad. Un detalle que merece la pena comentar es la estructura de la parte derecha de la figura 2.5. Este es uno de los pequeños detalles que hay que tener en cuenta cuando uno empieza a hacer sistemas cada vez más grandes: el orden. Sobre todo, si posteriormente hay otras personas que van a trabajar con él. Se ha intentado en todo momento mantener la comunicación entre los bloques mediante una señal multiplexada. Cuando no era posible, o esto complicaba más el diseñose han metido demultiplexores. Aquí se puede ver también como se ha separado la coordenada z con su respectiva velocidad de las otras dos, debido a que se han implementado en controladores distintos. La figura 2.6 muestra el detalle del bloque Demux, que consiste en un demultiplexor que selecciona una sola salida, la que nos interesa para un control de más alto nivel.

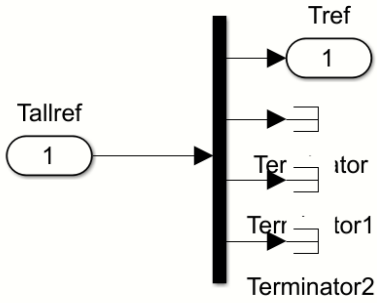

Figura 2.6 - Detalle de demultiplexor con una sola salida

Por último, si nos adentramos al último nivel (bloque QUAD + FFControl) dentro del simulador diseñado en Simulink se podrá observar lo que aparece en la figura 2.7. Se pueden observar tres cosas importantes: en primer lugar, las limitaciones en rad/s que tienen los motores (esta limitación ha sido elegida de manera arbitraria, con cierta coherencia con la realidad, pero podrían haber sido otras; de hecho, será otro parámetro para adaptar acorde con el modelo que queramos simular), el control de inversión y el quadrotor con el control de bajo nivel.

Como se puede observar, es un simulador pensado para verificar los controladores, pero al tener una arquitectura modular nos permite modificar cualquier control y probarlo en cuestión de segundos. Esta arquitectura se mantendrá a partir de ahora hasta el simulador final de Unity3d, que cambiará en parte, pero no adelantemos.

Vamos ahora a diseñar todas las funciones que rellenan todos estos bloques y que realmente son el núcleo del simulador. Hay que nombrar que todas las funciones se han simulado con untiempo de muestreo Tm como parámetro. Ya se hablará más delante de él, pero en definitiva se intenta emular el comportamiento real de una computadora de un dron que, por ser una computadora, es discreta.

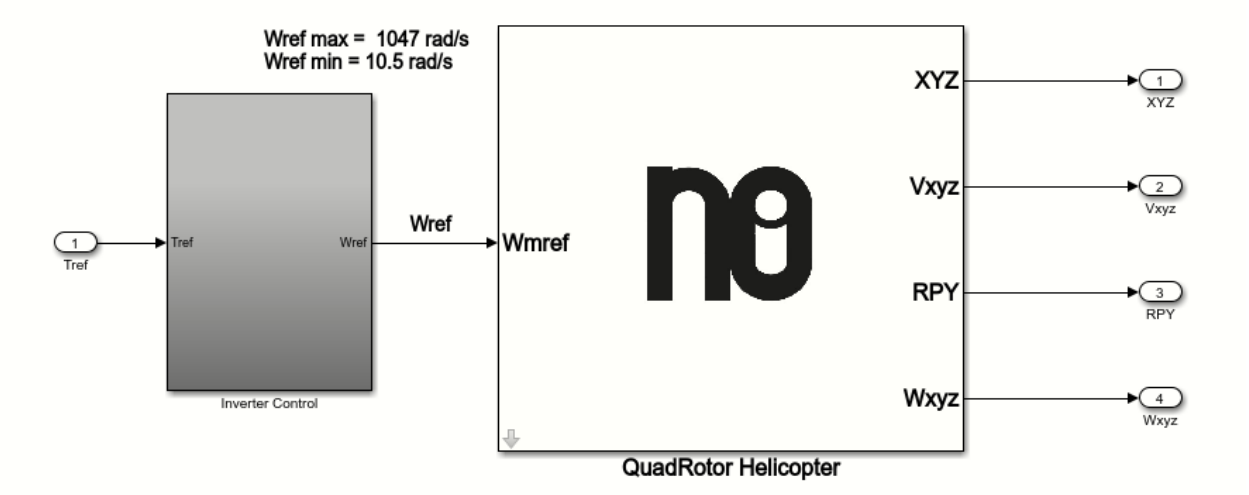

Figura 2.7 - Detalle del último nivel

2.2 Inserción de un modelo con un controlador de bajo nivel

En primer lugar cabe decir que el modelo utilizado en el simulador de Simulink ha sido prestado por el profesor Manuel Gil Ortega Linares. La justificación de utilizar este modelo y no uno propio es la finalidad de este simulador: comprobar el buen funcionamiento de los controladores, y no la creación de un simulador complete. Posteriormente se hablará del modelo dinámico y cinemático creado para el simulador final (el realizado en Unity3d).

Este modelo contiene todas las ecuaciones que gobiernan la cinemática y la dinámica del quadrotor. En la figura 2.8 se pueden observer el espacio de estados implementado.

$$\begin{pmatrix} \dot{\vec{P}}_B \\ {}^I\dot{\vec{V}}_B \\ \dot{\vec{\eta}} \\ \dot{\vec{\Omega}}_B \end{pmatrix} = \begin{pmatrix} {}^I\vec{V}_B \\ \frac{1}{m}{}^I R_B\left(\sum_i \vec{T}_{bi} + \sum \vec{F}_{Aerodynamics}\right) \\ W_\eta^{-1\,B}\vec{\Omega}_B \\ I^{-1}\left(\sum_i \vec{\tau}_{bi} + \sum \vec{\tau}_{Aerodynamics} - \vec{\Omega}_B \times I\vec{\Omega}_B\right) \end{pmatrix}$$

Figura 2.8 - Ecuaciones que gobiernan la dinámica dentro del modelo

A su vez, este modelo dado contiene una implementación de un control de bajo nivel que consigue que la velocidad de los motores giren a la velocidad de referencia dada.

Otra característica es que es un modelo balanceado y equilibrado. Esto implica varias cosas:

- El quadrotor posee simetría geométrica.
- El centro de masas se encuentra en el centro geométrico.
- Los cuatro motores son idénticos.

Estas características simplifican el diseño del control elegido.

Este modelo es una caja negra para el autor, por lo que no se puede saber más información de éste.

Antes de seguir, es importante aclarar el sistema de coordenadas escogido para este simulador. Esto ha sido un

gran dilema para este proyecto como se verá más adelante.

En la figura 2.8 se puede observar el utilizado para el simulador de Simulink [5]. Cuando más adelante se hable de los ángulos se hablarán de los ángulos Roll, Pitch y Yaw directamente. Al Roll también se le ha llamado alabeo; al pitch cabeceo y al yaw guiñada, como tradicionalmente se ha hecho.

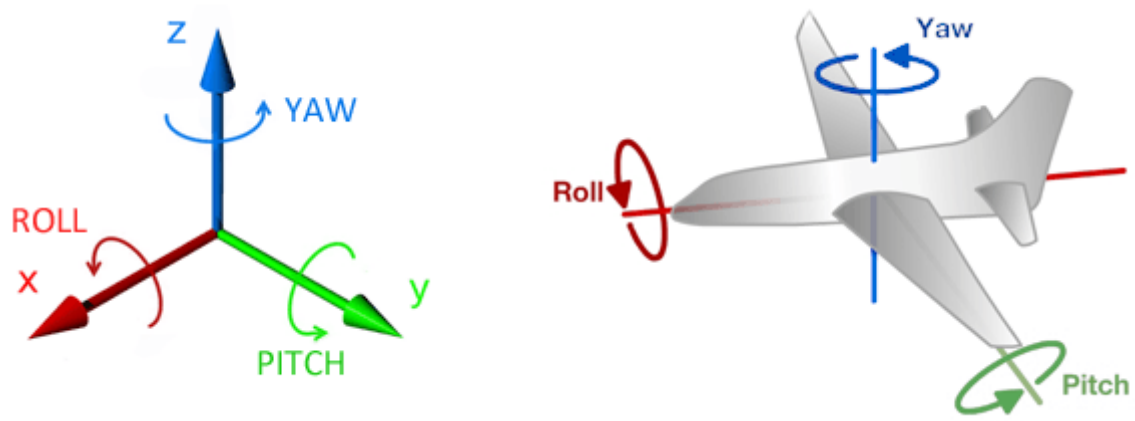

Figura 2.8 - Ángulos

Hay una serire de parámetros intrínsecos al quadrotor que definen, entre otras cosas, el empuje que realiza una hélice:

$$Masa(m) = 2.24Kg$$

$$Distancia\ de\ los\ ejes\ al\ centro\ de\ masas(L) = 0.332m$$

$$\text{Coeficiente de empuje}(b) = 9.5\text{e} - 6\ \text{N} * s^2$$

$$Coeficiente\ de\ arrastre(Kt) = 1.7e - 7\ N * m * s^2$$

$$Tensor\ de\ inercia(I) = [0.0363, 0.0363, 0.0615]Kg * m^2$$

La función que define al modelo se puede escribir de la siguiente manera:

$$[X, Y, Z, Vx, Vy, Vz, Roll, Pitch, Yaw, Wx, Wy, Wz] = Model(Wref1, Wref2, Wref3, Wref4)$$

2.3 Control de inversión

El control de inversion, como anteriormente se ha explicado, no es propiamente un control. Son una serie de algoritmos que componen una transformación que puede seguir la siguiente función:

$$[Wref1, Wref2, Wref3, Wref4] = Invert(Tref, Taurollref, Taupitchref, Tauyawref)$$

En primer lugar, entendamos el concepto de esta transformación. Hasta este punto, es decir, los controladores que existen de mayor nivel no entienden que el sistema que están controlando es un quadrotor. Cada uno controla unas variables, pero como si fueran variables independientes entre sí. Es en este paso donde el control tiene que transformar unos posibles cambios en la posición a los parámetros que podemos modificar en un quadrotor básico: las velocidades de los motores. Un ejemplo gráfico puede ser el siguiente: se quiere girar el ángulo roll 15º; el quad no entiende de ángulos, pero sí que entiende que si aumentamos la velocidad del 4º motor y disminuimos la del 2º, se girará. Para el pitch podríamos decir algo semejante, cambiando los motores al 1º y al 3º. Para el yaw se ha querido girar por exceso de velocidad, esto quiere decir que no se disminuye la velocidad de ningún motor implícitamente, sino que se aumenta la velocidad de los motores pares o impares, dependiendo de si se quiere girar en un sentido u otro. Para la altura es la más sencilla dado que incidimos directamente en los cuatro motores al mismo tiempo. Con esta vista es fácil intuir que la velocidad de referencia de cada motor será la suma de todas las velocidades dichas anteriormente. Un detalle que se verá más adelante, pero que puede ayudar ahora más, es que el par virtual de la altura (Tref) es el único que no oscila alrededor del 0, sino que lo hace alrededor de Teq, que es el valor correspondiente a Weq que es la

velocidad exacta que consigue contrarrestar la fuerza del peso del quadrotor.

MODELO DE GENERACIÓN INVERTIDO

```
FF = [ 1/4,         0, -1/(2*L),  b/(4*Kt);...
       1/4,  1/(2*L),         0, -b/(4*Kt);...
       1/4,         0,  1/(2*L),  b/(4*Kt);...
       1/4, -1/(2*L),         0, -b/(4*Kt) ];
Fref = FF*[Tref; Taurollref; Taupitchref; Tauyawref];
wref = sqrt(1/b * Fref);
```

Como se puede observar en el código anterior, muestra analíticamente lo dicho de manera funcional. Hay varios parámetros intrínsecos al quadrotor explicados en el modelo.

También se puede observar la inversión de velocidades entre los pares y lo impares.

Por último, se han saturado las velocidades de referencia para cada motor:

SATURACIONES

```
Wref = [wref(1); wref(2); wref(3); wref(4)];
wmax = 1047; % (rad/s)
wmin = 10.5; % (rad/s)
for i = 1:4
    if Wref(i) > wmax
        Wref(i) = wmax;
    elseif Wref(i) < wmin
        Wref(i) = wmin;
    end
end
```

2.4 Control de estabilidad

El control de estabilidad es el encargado de mantener el quadrotor en una posición estable en régimen permanente. Se encarga de controlar los tres ángulos y la altura (en este caso el eje Z). La función que define a este control es la siguiente:

$$[Tref, Taurollref, Taupitchref, Tauyawref] = STControl(ZRPYref, ZRPY, Vz, Omega[3], t)$$

Donde ZRPY es Z, Roll, Pitch y Yaw, en forma de referencia o medida real dependiendo del sufijo que le acompañe.

En primer lugar, veremos el código y posteriormente veremos el diseño de todos los controladores.

El código del control de estabilización tiene dos partes grandes:

1. La primera consta de una serie de fórmulas que tratan de saber las velocidades de cada ángulo, debido a que el modelo supone que los ejes permanecen fijos al recibir un giro en cualquier eje. En segundo lugar, también se implementa una derivada numérica para obtener la velocidad de referencia en cada ángulo y en la altura. Al incluir las velocidades de referencia en este apartado conseguimos implementar un pequeño control sobre las velocidades. Posteriormente, este diseño cambiará.

Obtención de las velocidades de los ángulos

```
Wn_inv = [1, sin(Roll)*tan(Pitch), cos(Roll)*tan(Pitch);...
          0, cos(Roll)           , -sin(Roll);...
          0, sin(Roll)*sec(Pitch), cos(Roll)*sec(Pitch)];
Angd = Wn_inv*Omega';
Rolld = Angd(1);    Pitchd = Angd(2);   Yawd = Angd(3);
```

Implementacion de una derivada numerica para la velocidad de la referencia

```
Rolldref  = (Rollref - Rollref1) / Tm;
Pitchdref = (Pitchref - Pitchref1) / Tm;
Yawdref   = (Yawref - Yawref1) / Tm;
Vzref     = (Zref - Zref1) / Tm;
```

Saturamos las referencias en velocidad

```
conver = pi/180; % Conversion de grados a radianes
wangmax = 10*conver;   % (rad/s)
wangmin = -10*conver;  % (rad/s)
vzmax = 0.5;          % (m/s)
vzmin = -0.5;         % (m/s)
if Rolldref > wangmax
    Rolldref = wangmax;
elseif Rolldref < wangmin
    Rolldref = wangmin;
end
if Pitchdref > wangmax
    Pitchdref = wangmax;
elseif Pitchdref < wangmin
    Pitchdref = wangmin;
end
if Yawdref > wangmax
    Yawdref = wangmax;
elseif Yawdref < wangmin
    Yawdref = wangmin;
end
if Vzref > vzmax
    Vzref = vzmax;
elseif Vzref < vzmin
    Vzref = vzmin;
End
```

2. La segunda parte es donde se implementan los cuatro controladores. Es aquí donde se puede percibir el detalle nombrado anteriormente: el único controlador que tiene un punto de equilibrio distinto a 0 es el control de altura. El control de altura se ha implementado en este controlador, pudiéndose implementar en el de posición siguiendo el orden coherente con el nombre del controlador. Al final se implementado en este nivel con la intención de reflejar aquellos controles que tienen el mismo número de niveles debajo de ellos. A diferencia de las coordenadas X e Y que tienen que actuar en los ángulos Roll y Pitch, la coordenada Z no tiene por debajo ningún ángulo a controlar. El código se muestra a continuación:

IMPLEMENTACIÓN DEL PD DE ALTURA (Z)

```
ez = Zref - Z;          % Error en posición
evz = Vzref - Vz;       % Error en velocidad
intez = intez + ez;     % Integral del error de la altura
Ktf = 1/(cos(Roll)*cos(Pitch));
Kph = 45.4109; Kdh = 0.3587*Kph;
Tref = Teq + Ktf*(Kph*ez + Kdh*evz);
```

IMPLEMENTACIÓN DEL PD DEL ALABEO (ROLL)

```
eroll = Rollref - Roll;         % Error en angulo
erolld = Rolldref - Rolld;      % Error en velocidad
```

```
Kproll = 0.6815; Kdroll = 0.3766*Kproll;
Taurollref = Kproll*eroll + Kdroll*erolld;
```

IMPLEMENTACIÓN DEL PD DEL CABECEO (PITCH)

```
epitch = Pitchref - Pitch;        % Error en angulo
epitchd = Pitchdref - Pitchd;     % Error en velocidad
Kppitch = Kproll; Kdpitch = Kdroll;
Taupitchref = Kppitch*epitch + Kdpitch*epitchd;
```

IMPLEMENTACIÓN DEL PD DE LA GUIÑADA (YAW)

```
eyaw = Yawref - Yaw;              % Error en angulo
eyawd = Yawdref - Yawd;           % Error en velocidad
Kpyaw = 1.6383; Kdyaw = 0.3022*Kpyaw;
Tauyawref = Kpyaw*eyaw + Kdyaw*eyawd;
```

2.5 Control de posición

El control de posición se encarga de ir controlando las referencias de los ángulos en el plano X e Y para que el quadrotor se acerque a la posición deseada en este plano. La función que define a este control es la siguiente:

$$[Rollref, Pitchref] = PosControl(XYref, XY, Vxy, Yaw, Tref, t)$$

Al igual que en el caso anterior, veremos antes el código y posteriormente habrá una sección dedicada únicamente al diseño de los controladores.

Este código también incluye dos partes distinguibles, con semejanza al anterior, pero puestas de manera inversa, es decir, primero controlamos en XY y posteriormente aplicamos la transformación requerida:

1. Implementación de los controles para las coordenadas X e Y. Para ello se ha necesitado implementar una derivada numérica para las velocidades de referencia, teniendo en cuenta el mismo esquema del control de estabilidad. Un detalle importante es que al estar tratando implementación discreta, el Tref utilizado (recordar que el Tref es el par virtual aplicado a los cuatro motores para mantener la altura) es el del instante anterior:

IMPLEMENTACION DERIVADA NUMÉRICA

```
Vxref = (Xref - Xref1) / Tm;
Vyref = (Yref - Yref1) / Tm;
```

IMPLEMENTACION DEL PD EN X

```
ex = Xref - X;          % Error en posicion
evx = Vxref - Vx;       % Error en velocidad
Kpx = 1.7105*2; Kdx = 2.2568*0.5*Kpx;
Uxref = (Kpx*ex + Kdx*evx) / Tref;
```

IMPLEMENTACION DEL PD EN Y

```
ey = Yref - Y;           % Error en posicion
evy = Vyref - Vy;        % Error en velocidad
Kpy = Kpx; Kdy = Kdx;    % Simetrías
Uyref = (Kpy*ey + Kdy*evy) / Tref;
```

2. La segunda parte transforma Uxref e Uyref en Rollref y Pitchref. Por último, saturamos las referencias

de estos ángulos, a un máximo de 15º en ambos sentidos, debido que estos dos ángulos son la principal causa de inestabilidad:

TRANSFORMACIÓN (Uxyref to RPref)

```
Rollref = asin(sin(Yaw)*Uxref - cos(Yaw)*Uyref);
Pitchref = asin((cos(Yaw)*Uxref + sin(Yaw)*Uyref) / (cos(Rollref)));
```

SATURACIONES Y OBTENEMOS LA SALIDA

```
conver = pi/180; % Conversion de grados a radianes
rollmax = 15*conver;   % (rad)
rollmin = -15*conver;  % (rad)
pitchmax = 15*conver;  % (rad)
pitchmin = -15*conver; % (rad)

if Rollref > rollmax
    Rollref = rollmax;
elseif Rollref < rollmin
    Rollref = rollmin;
end

if Pitchref > pitchmax
    Pitchref = pitchmax;
elseif Pitchref < pitchmin
    Pitchref = pitchmin;
End
```

2.6 Generadores de trayectorias: lineal y Smart

Para realizar todas las pruebas quedaba implementar un pequeño algoritmo que consiga suavizar las referencias de los 4 grados de libertad: X, Y, Z, Yaw. Recuerdo: Roll y Pitch deben estar a 0º en regimen permanente. Por tanto el generador debería seguir la siguiente función:

$$[Xref, Yref, Zref, Yawref] = GT(Pfinal, Pinicial)$$

Donde Pfinal es el punto objetivo y Pinicial es el punto inicial de la trayectoria (con normalidad donde se encuentra actualmente el quadrotor). Se han añadido dos parámetros más para aumentar las prestaciones de este GT (generador de trayectorias):

- Tmov: tiempo en el que el movimiento se va a realizar.
- Tinicial: instante en el cual se iniciará el movimiento.

Ambos tiempos están en segundos. Finalmente, la función queda como sigue:

$$[Xref, Yref, Zref, Yawref] = GT(Pfinal, Pinicial, Tmov, Tinicial)$$

Se han dasarrollado don generadores de Splits para unir el punto final y el inicial: lineal y Smart.

2.6.1 Generador lineal

Este generador es el generador más básico. Se trata de un generador que es doblemente lineal, es decir, es geométricamente y temporalmente lineal. Esto implica que la posición se verá afectada linealmente, la velocidad será constante y la aceleración nula (salvo en el instante final e inicial).

Se puede observar esta doble linealidad en el código del GT:

Generador de Splits lineal

```
if (tiempo < t_ini)
```

```
        Xout = 0;
        Yout = 0;
        Zout = 0;
    elseif (tiempo < t_mov + t_ini)
        Xout = (tiempo - t_ini)/t_mov * Xref;
        Yout = (tiempo - t_ini)/t_mov * Yref;
        Zout = (tiempo - t_ini)/t_mov * Zref;
    elseif (tiempo >= t_mov + t_ini)
        Xout = Xref;
        Yout = Yref;
        Zout = Zref;
    end
```

Salidas

```
    refs = [Xout; Yout; Zout; Yawref];
```

Dado que no se buscaba encontrar el mejor generador, se utilizó este generador para el simulador final de Unity3d. Una vez probado se comprobó que es muy mejorable. En concreto se observe que, al tener una aceleración grande al inicio y al final, debido al escalón en velocidad que conlleva este generador, sufría grandes perturbaciones el quad. Se optó por mejorar la velocidad y aceleración, dejando igual la posición. Así se dio lugar al generador Smart.

2.6.2 Generador Smart

Se trata de un generador que es lineal geométricamente, pero temporalmente sigmoidal. Para ver una demostración gráfica del funcionamiento se puede observar la figura 2.9. En ella se puede observar como la trayectoria final (o el split en este caso, porque es para cada instante), es una recta en el espacio 3d. Pero viando la parte derecha de la figura 2.9 se puede observar que ya no es una línea recta con respecto al tiempo.

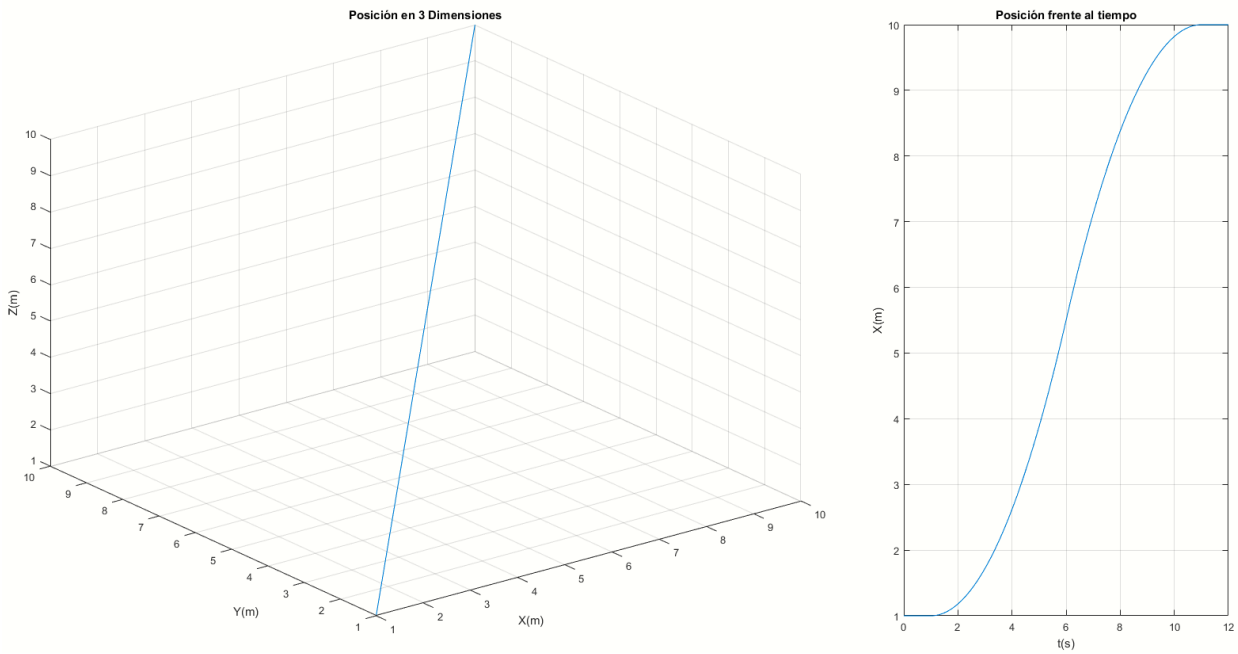

Figura 2.9 - Salida generador Smart

Es fácil comprender con estas dos gráficas que la velocidad será máxima justo en el instante central de la trayectoria, describiendo una parabola invertida. La aceleración no es trivial verla, pero se puede intuir que es una cúbica con el punto de infexión donde la velocidad es máxima.

Con este generador mejora notablemente el comportamiento del quadrotor; y por eso mismo fue este

finalmente el que se utilizó en el Simulador final de Unity3d. En el simulador de Simulink, pensado para pruebas de los controladores, no se llegó a introducer este generador.

El código se muestra a continuación:

Código generador Smart

```
Pini = [1, 1, 1, pi/2];
Pfin = [10, 10, 10, pi];
t_mov = 10;
t_ini = 1;

t = [0:0.01:12];
Punto = zeros(4, length(t));
a = t_ini;
c = t_ini + t_mov;
Obj = Pfin - Pini;
Ini = Pini;

for i = 1:length(t)
    % Obtenemos los puntos intermedios
    Punto(:,i) = Pini';
    if (t(i) > t_mov + t_ini) % Tramo final
        Punto(:,i) = Pfin;
    elseif (t(i) > (t_mov/2 + t_ini))
        Punto(:,i) = (1 - 2*((t(i) - c)/(c - a))^2) * Obj + Ini;
    elseif (t(i) > t_ini)
        Punto(:,i) = 2*((t(i) - a)/(c - a))^2 * Obj + Ini;
    end
end
```

El código no es una función, sino que se implementó un script únicamente para ver su funcionamiento. Se puede observar que se compone de dos funciones que enganchan justo a mitad de tiempo. La segunda es la inversa en X e Y de la primera (desplazada convenientemente).

2.7 Diseño de los controladores

En total se ha llevado a cabo el diseño de seis controladores, respecto a los seis grados de libertad (tres de posición más tres de ángulos).

El diseño de los controladores ha sido en orden de menor nivel a mayor. Se ha partido de la siguiente función de transferencia para el modelo:

$$G(s) = \frac{w(s)}{wref(s)} = \frac{1}{(0.04s + 1) * (0.04s + 1)}$$

Partiendo de esta base se ha diseñado el siguiente nivel (control de estabilidad, dado que el de inversión es instantáneo) pensando en controlar esa G(s) añadiendo los términos correspondientes. Empezando por el caso del ángulo Roll, obtenemos la siguiente Groll(s), que incluye el nivel anterior (modelo más control de bajo nivel) más el término inercial corresponiente:

$$\text{Groll}(s) = \frac{1}{Ixx * s^2 * (0.04s + 1) * (0.04s + 1)}$$

Así, por el método del lugar de las raíces se ha diseñado el siguiente PD, por ser el más fiable, contando con poca precisión en el diseño:

$$Croll(s) = 0.25668 * (s + 2.655)$$

Posteriormente se ha discretizado con un tiempo de muestro de 0.001s, utilizando la aproximación de Euler II.

Al final queda como se ha podido observar ya en el código del control de estabilidad:

$$Taurollref = Kproll * eroll + Kdroll * erolld$$

Siendo Kproll = 0.6815 y Kdroll = 0.3766*Kproll. Referir la Kd a la Kp es una manera de mantener la proporcionalidad entre las aportaciones del proporcional y el derivativo cuando se maneja el tiempo de respuesta en bucle cerrado; es simplemente una ayuda.

De la misma manera se han diseñado los demás controladores de Pitch y de Yaw, resultando:

$$\text{Gpitch}(s) = \frac{1}{Iyy * s^2 * (0.04s + 1) * (0.04s + 1)}$$

$$Cpitch(s) = 0.25668 * (s + 2.655)$$

$$\text{Gyaw}(s) = \frac{1}{Izz * s^2 * (0.04s + 1) * (0.04s + 1)}$$

$$Cyaw(s) = 0.4951 * (s + 3.309)$$

Como se puede observar, el pitch es idéntico al roll, debido al equilibrio de partida del modelo. El control en altura es similar, debido a que está al mismo nivel:

$$\text{Gz}(s) = \frac{1}{m * s^2 * (0.04s + 1) * (0.04s + 1)}$$

$$Cz(s) = 16.288 * (s + 2.788)$$

Como se puede observar, la ganancia del controlador de la altura tiene una ganancia muy alta. Esto ha sido una decisión del diseño. Debido a que es un controlador que influye directamente en los cuatro motores de manera simultánea, es el que, sufriendo grandes cambios, menos desestabiliza el sistema. Esto abre una posible fuente para implementar un evasor de obstáculos rápido tocando fundamentalmente la altura.

Para los controladores en las coordenadas X e Y, que ya están en un nivel superior, se ha supuesto que tienen que controlar un sistema diez veces más lento que los anteriores:

$$\text{Gx}(s) = \frac{1}{m * s^2 * (0.4s + 1) * (0.04s + 1)}$$

$$\text{Gy}(s) = \frac{1}{m * s^2 * (0.4s + 1) * (0.04s + 1)} = Gx(s)$$

También sigue permaneciendo la simetría en las coordenadas. El control resultante ha sido:

$$Cx(s) = 3.8604(s + 0.4431)$$

$$Cy(s) = Cx(s)$$

Posteriormente, al empezar a conocer el entorno de Unity3D, se vio necesario aumentar el tiempo de muestro hasta 20 veces mayor:

$$\text{Tm} = 0.02\ s$$

Este problema se explicará más adelante, pero la causa es si no se aumentaba el tiempo de muestro, el ordenador era incapaz de correr el programa. Esto no fue un problema, primero porque desde el opunto de vista de la programación no afecta, dado que todos los controladores están dependientes del Tm. En segundo lugar, se pensó que esto podría influir significativamente al funcionamiento de los controladores, dado que, si se aumenta demasiado el Tm, el sistema en bucle cerrado puede volverse inestable. Como se verá en el siguiente apartado, esto no ha supuesto ningún problema.

2.8 Simulación

Se han realizado multitud de simulaciones, con idénticos resultados. A continuación, se muestran los resultados de una simulación con los siguientes parámetros:

- Se ha utilizado el generador lineal.
- Posición inicial: se ha partido del origen [0, 0, 0, 0], para no añadir al generador más parámetros, dado que en el futuro se desechará. Las posiciones requieren este formato [X, Y, Z, Yaw].
- Posición final: posición objetivo en [-10 m, -10 m, 10 m, - 80 º];
- Tiempo de movimiento: 20 s.
- Tiempo inicial del movimiento: 5 s, desde el comienzo de la simulación.
- Tiempo de simulación: 40 s.
- Tiempo de muestro: 0.02s. Se ha considerado que la peor simulación, y la más necesaria, debido a que es en la que nos basaremos en el futuro, es con este tiempo de muestro. Quede por entendido, que disminuyendo el tiempo de muestro mejoran los resultados.

Los resultados han demostrado la bondad del sistema controlado por 4 niveles en cascada.

Como se puede observar en la figura 2.10 y 2.11 (si se fija la mirada en el ángulo Yaw), obtenemos un error constante durante toda la duración del movimiento, debido a que el sistema en bucle cerrado solo tiene un integrador. Esto causa que, ante referencia en forma de rampa, como es el caso del generador lineal, el error sea constante. Una vez terminado el régimen permanente, el sistema percibe una referencia en forma de escalón, por lo que es capaz de alcanzar un error nulo, como se aprecia en la figura 2.12.

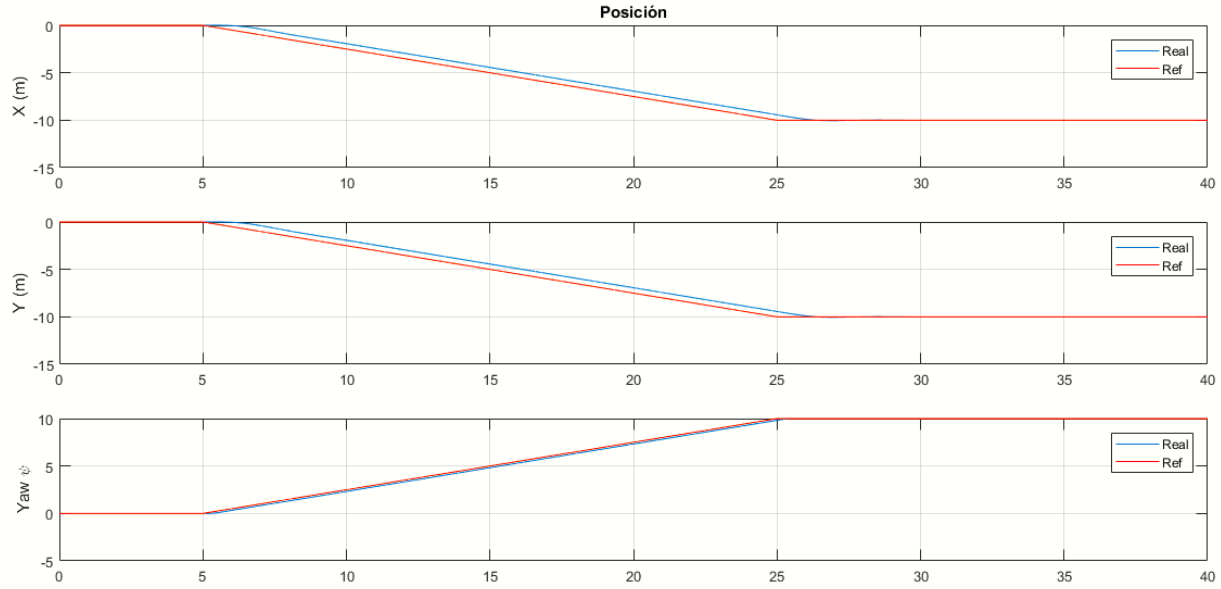

Figura 2.10 - Resultado de la simulación en términos de posición

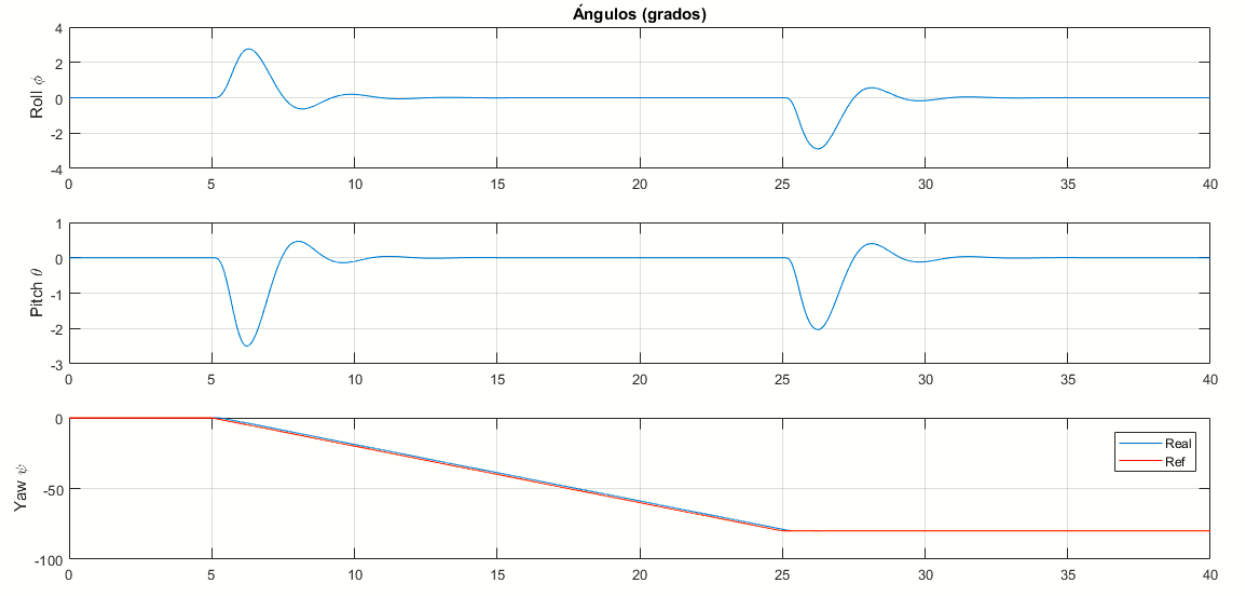

Figura 2.11 - Resultado de la simulación en términos de ángulos

Mirando la gráfica 2.13, podemos corroborar lo dicho anteriormente. Por último, observando las figuras 2.14 y 2.15, referidas a las velocidades, podemos también comprobar cómo la aceleración en los intantes inicial y final es enorme, frente a los instantes intermedios donde las velocidades permanecen constantes, la aceleración es nula. Este es el motivo de creación del generador Smart y su futura implementación en el simulador definitivo en la plataforma Unity3D.

Con esta simulación se da por comprobado la bondad de los controladores, su arquitectura en cascada y su implementación discreta con un tiempo de muestreo de **0.02s**.

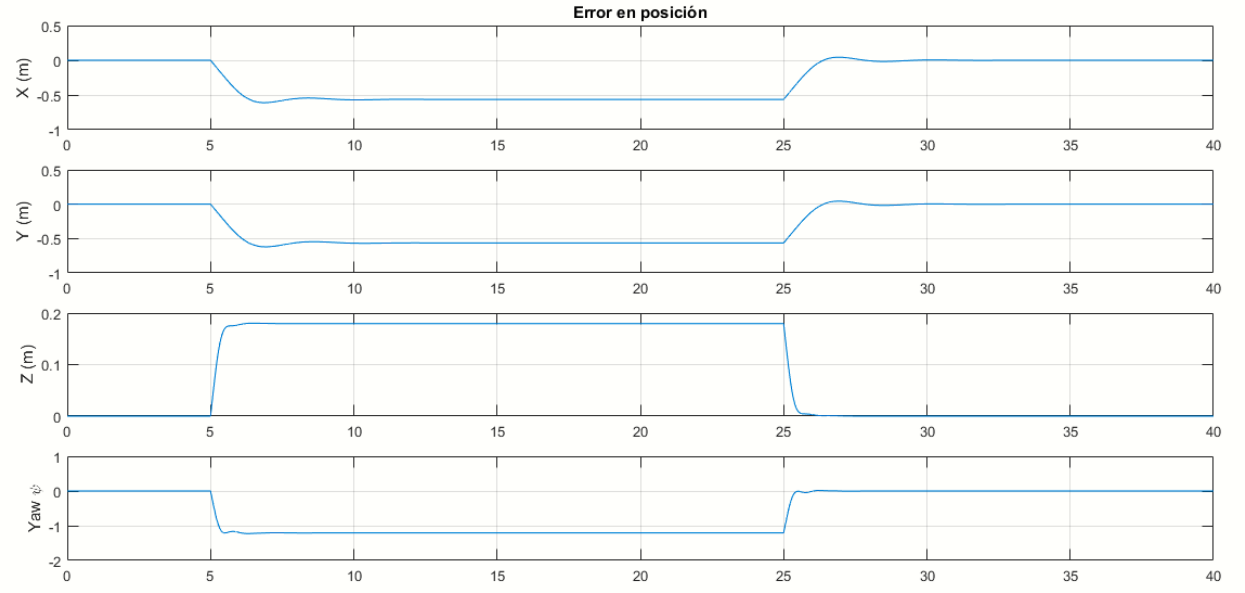

Figura 2.12 - Resultado de la simulación en términos de error en la posición completa

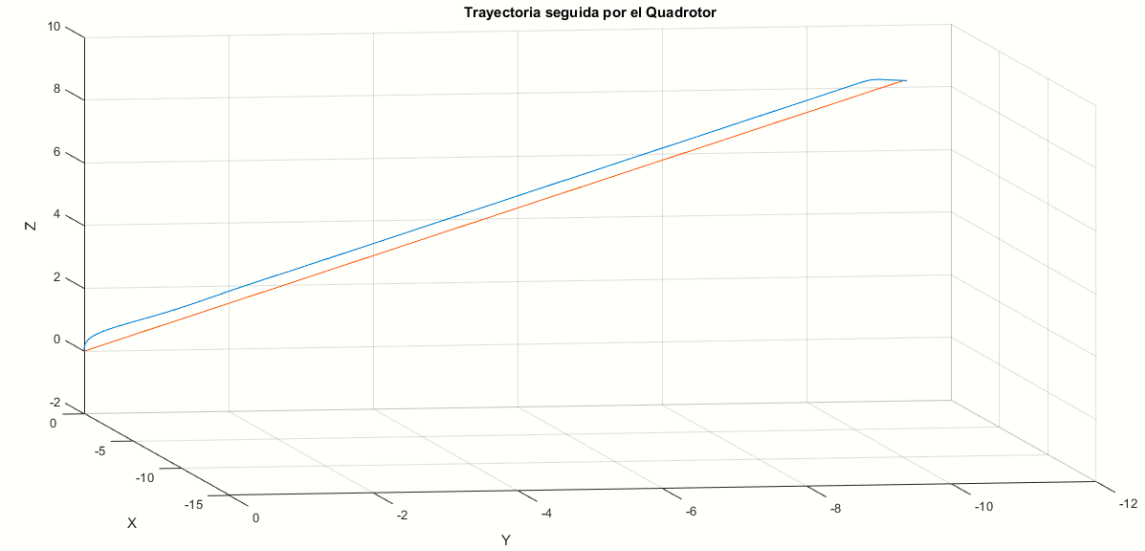

Figura 2.13 - Resultado de la simulación, representación en 3D del movimiento realizado

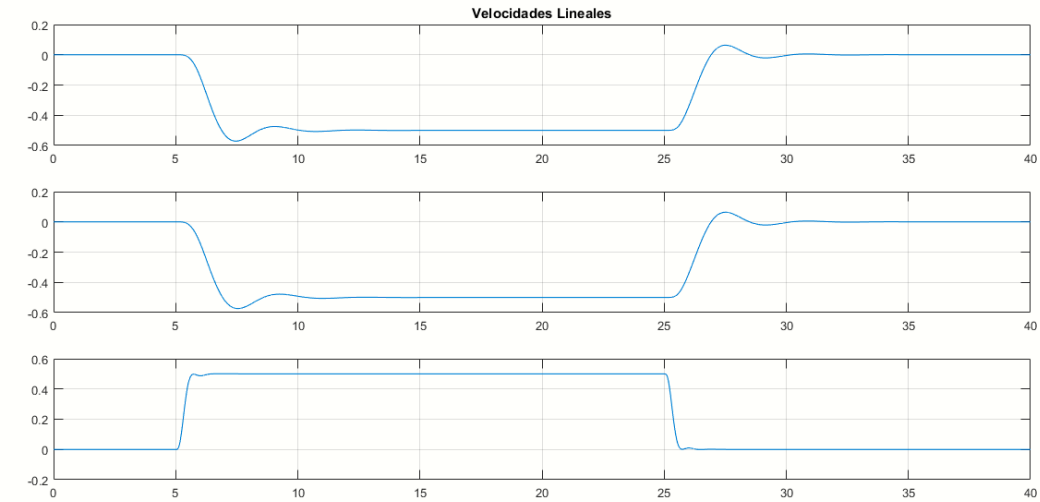

Figura 2.14 - Resultado de la simulación en términos de velocidad lineal

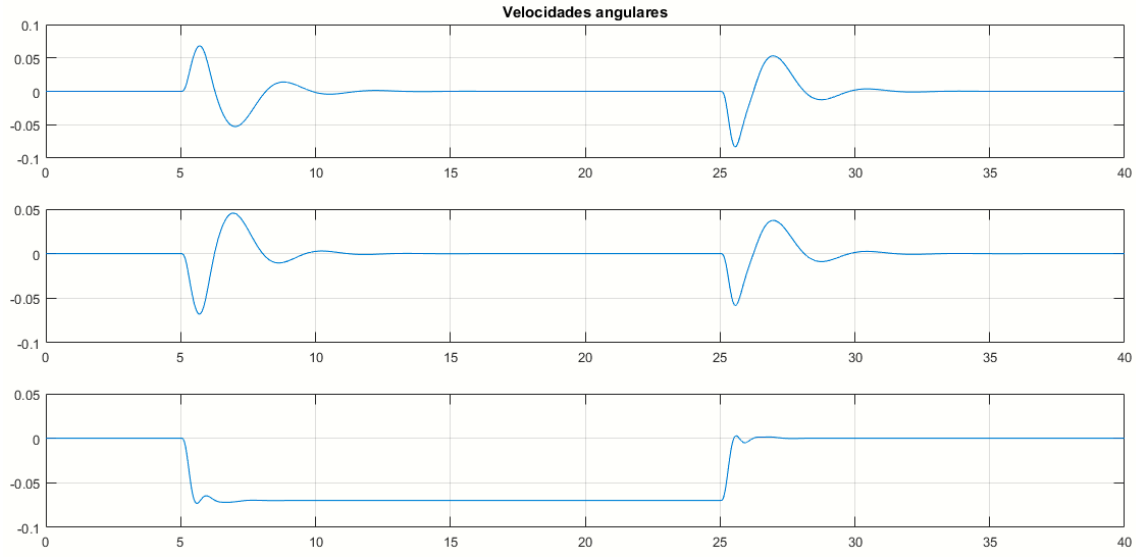

Figura 2.15 - Resultado de la simulación en términos de velocidad angular

Una vez verificado el funcionamiento del sistema, el siguiente paso es diseñar en 3D el quadrotor.

3 CREACIÓN DEL MODELO EN 3D

Es intentando lo imposible como se realiza lo posible

Henri Barbusse

En este capítulo vamos a describir el mundo del diseño en 3D dentro de uno de los programas más conocidos como puede ser el Cinema4D de Maxon [6]. Este programa tiene unas potencias increíbles. Se pueden realizar diseños en 3D con una semejanza hacia la realidad increíbles.

Podemos dividir el quadrotor diseñado en 4 grupos distintos:

1. Diseño del cuerpo completo.
2. Diseño de las torres motoras.
3. Diseño de la cámara.
4. Diseño de las hélices.
5. Personalización con la marca nO.
6. Adición de texturas.
7. Preparación para Unity3d.

Se irá explicando como se ha desarrollado cada parte con imágenes.

3.1 Diseño del cuerpo completo del Quadrotor

En primer lugar se hizo un diseño a mano. Este diseño buscaba dos propiedades importantes:

- Formación de un cuadrado entre los 4 ejes de las hélices. Con esto se quería justificar el futuro modelo que se iba a implementar en Unity3D.
- Ergonomía: se buscaba que el dron fuse una sola pieza donde ahí fuesen engarzando los elementos externos.

El preámbulo de diseño de esta parte más importante es que se diseñaría todo partiendo de una simetría. Esto implica que solo hay que diseñar la mitad del quadrotor. En la figura 3.1 se puede observar como quedó el diseño final del cuerpo sin aplicar la simetría, y mostrando las cuatro vistas del cuerpo.

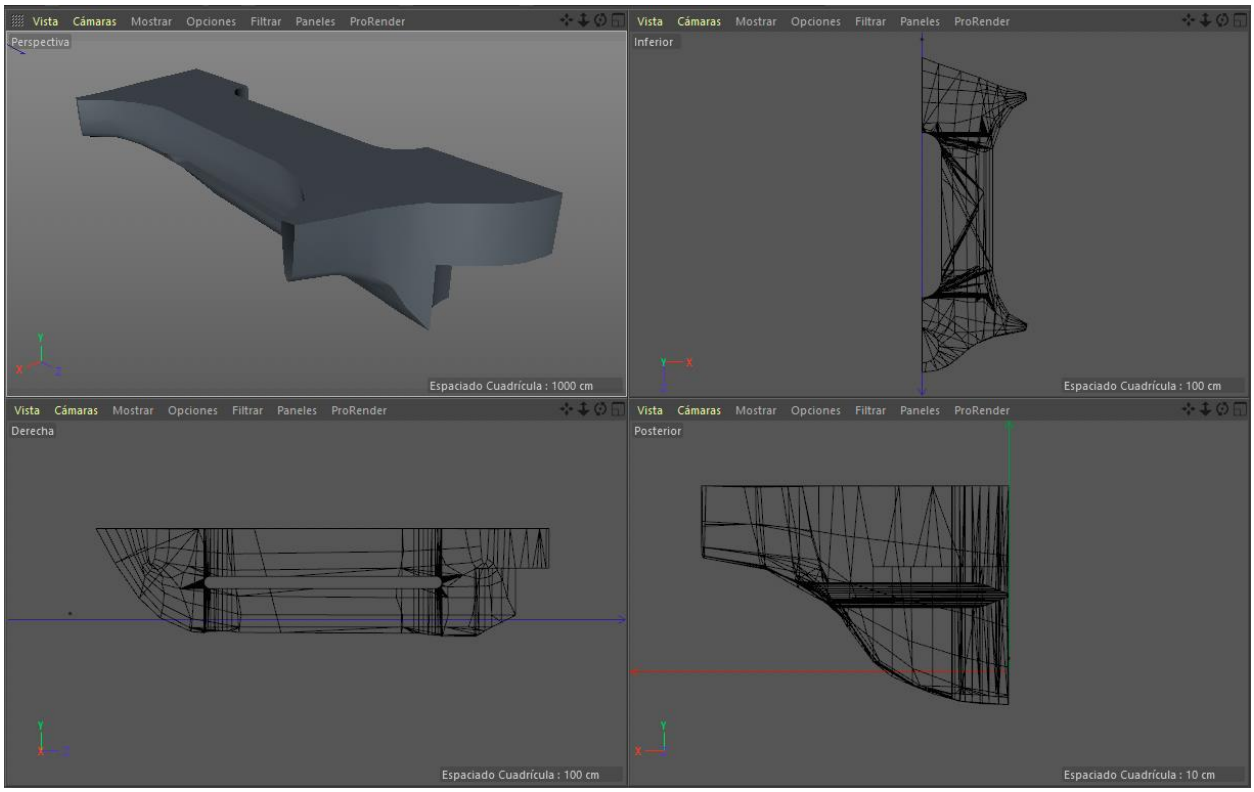

Figura 3.1 - Vistas del cuerpo

Los pasos seguidos para realizar el cuerpo han sido los siguientes:

1. Creación de un cubo en 3d.
2. Se le ha quitado la parte frontal con una “s” para que tome la forma de ola.
3. Con un ciclindro se ha rebajado el cubo para crear el espacio de la cámara (en la figura 3.2 se puede observar un detalle del hueco).
4. Mediante cortes y divisiones se ha realizado a mano el resto del diseño.
5. Después se han alargado las extensiones para separar las torres motoras, como puede observarse en la figura 3.2.
6. Por último, se le ha aplicado la simetría con respecto al eje x para obtener el cuerpo completo. Mírense las figuras 3.3 y 3.4.

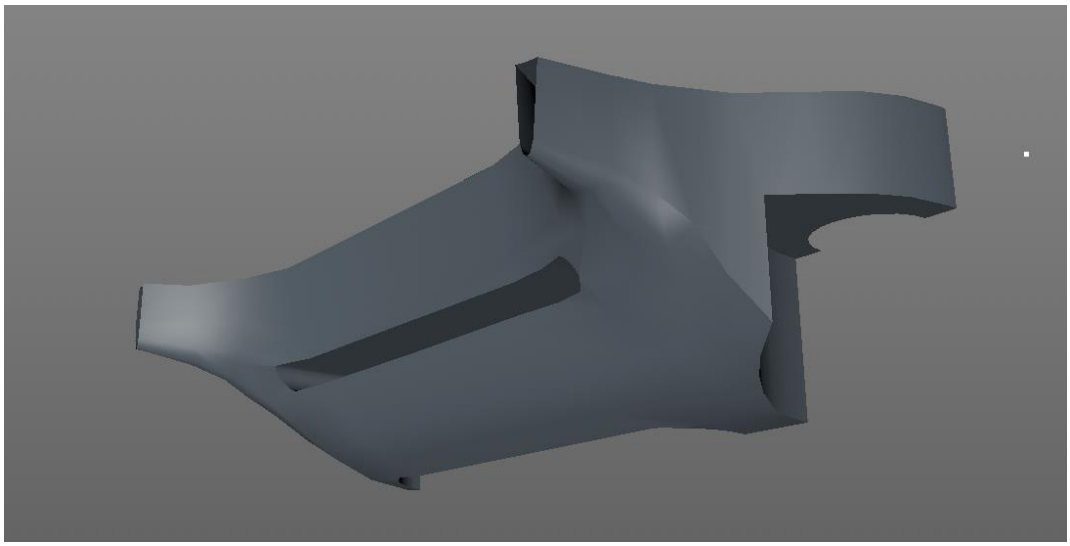

Figura 3.2 - Otra perspectiva del cuerpo sin simetría

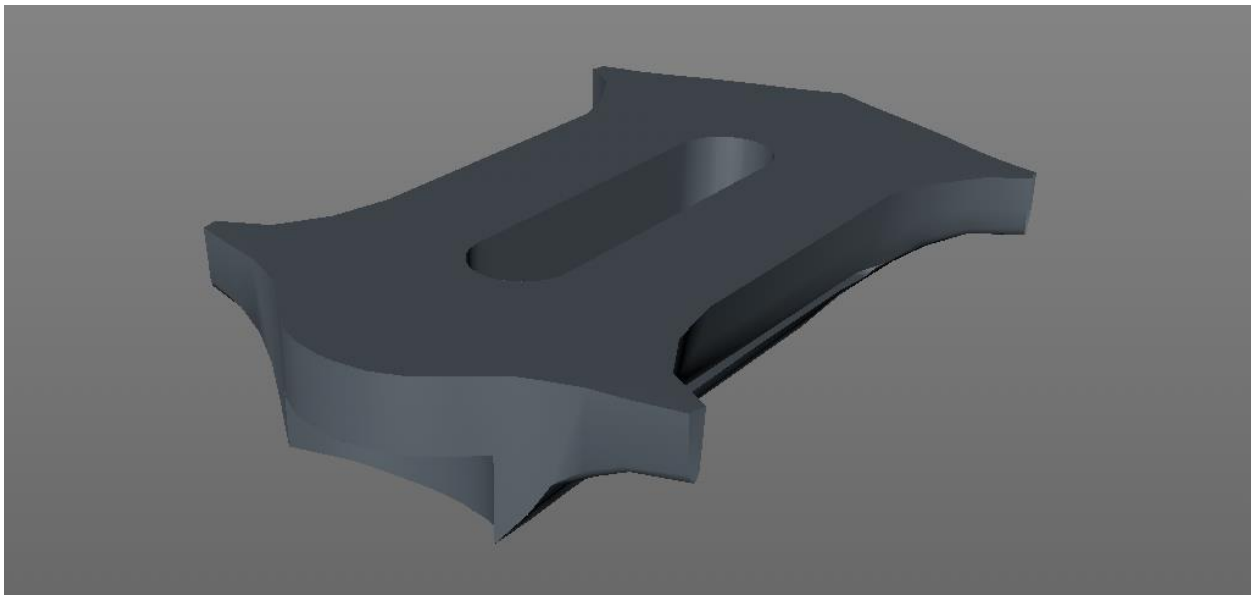

Figura 3.3 - Cuerpo con simetría aplicada

Una vez acabado el diseño, se pensó por estética, introducir dos huecos en los planos horizontal y vertical, transversales al cuerpo. Estos huecos que físicamente aumentaban la aerodinámica, hacían honor a unas de las ampliaciones pensadas para el proyecto: introducirlo en las gafas de realidad virtual Oculus Quest. Si miramos la figura 3.4, que nos muestra una vista superior del cuerpo podremos visualizar una semejanza con el logo de Oculus.

Figura 3.4 - Cuerpo con simetría aplicada visto desde arriba

Una vez diseñado el cuerpo, se ha pasado a diseñar un soporte para la cámara. Como se puede observar en la figura 3.5, consta de tres bases. Se pensó así, para emular el comportamiento con rigidez. Si fijamos la vista en la figura 3.6, podremos ver el soporte incluido en la simetría del cuerpo.

Figura 3.5 - Soporte sin simetría

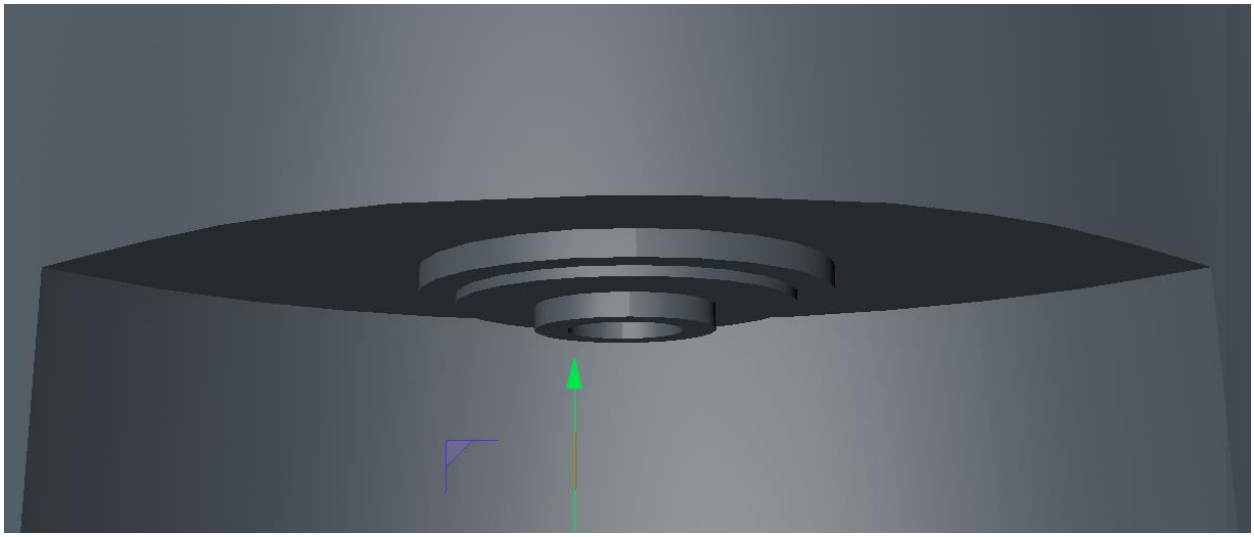

Figura 3.6 - Soporte con geometría

Posteriormente se ha diseñado un cristal de protección para la cámara. El diseño de la forma ha consistido en alargar todas las líneas del quadrotor para que se consiguiera una ergonomía completa. Si nos fijamos en la figura 3.7, podremos observar como la parte frontal viene dada por esa extracción en forma de "s" nombrada anteriormente, mientras que la parte inferior viene dada por el alarmgamiento de la cubierta inferior del quadrotor. En la figura 3.8 se puede observar la adición del cristal dentro de la simetría del cuerpo.

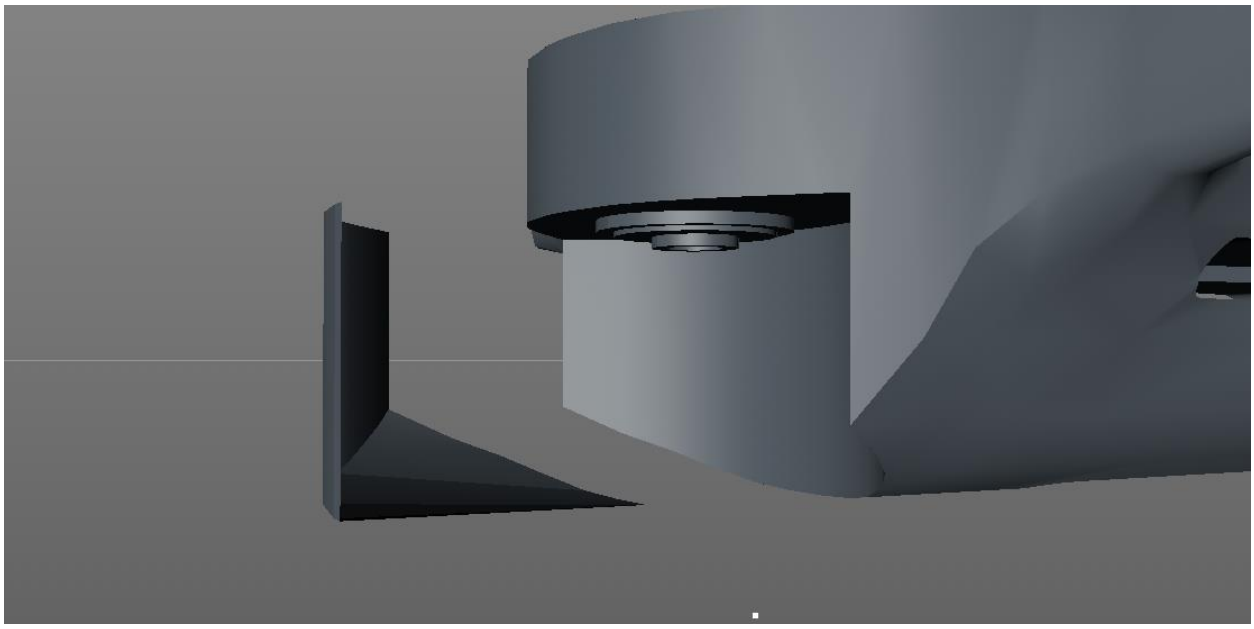

Figura 3.7 - Cristal de protección

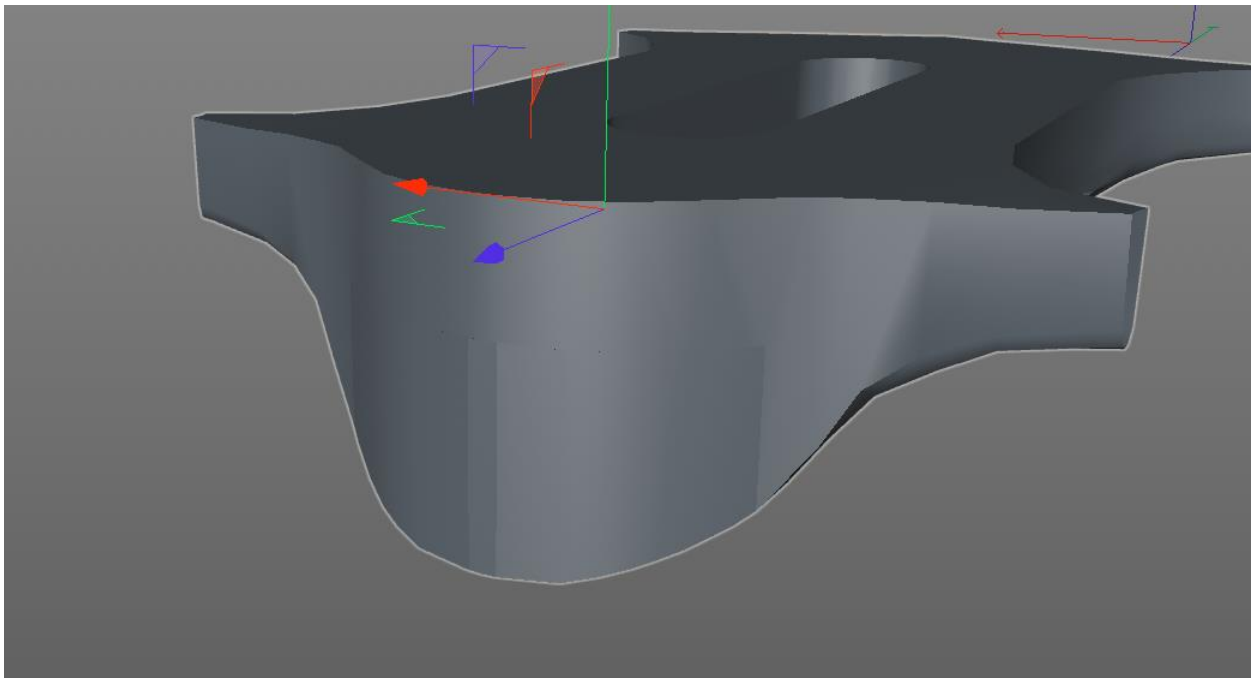

Figura 3.7 - Cristal de protección unido al cuerpo con simetría

3.2 Diseño de las torres motoras

Las torres motoras son las cuatro torres que por un lado mantienen fuera de contacto el cuerpo con el suelo y por otro lado son aquellas que tienen el motor incluído dentro.

Estéticamente hubo varias inciativas, como por ejemplo que fueran gruesas. Al final se fueron unificando todos los diseños hasta llegar al diseño final que se muestra en la Figura 3.8. Este diseño consta de los siguientes pasos:

1. En primer lugar se hecho un split, que es una línea, del contorno de la torre.
2. Se ha girado esa línea entorno al eje de revolución. Así se consigue la forma externa de la torre.
3. Depsués se ha diseñado la parte superior de la torreta. Todo lo que viene a continuación se ha hecho

de la misma manera: creando subdivisions concéntricas del plano superior. La primera de ellas ha sido crear un borde, elevarlo y girarlo, con eso se ha conseguido que la parte superior tenga ese contorno de protección para el motor.

4. A continuación se ha diseñado el motor (mírese la figura 3.9): se compone de 5 subdivisiones. La primera es el cilindro que representa al motor. La segunda y tercera son los dos huecos simétricos que representan como una vision interna del motor. Por último, la cuarta y quinta representan la sujección del eje de la hélice y el hueco para éste.
5. Por último se ha dejado plano la conexión con el cuerpo (se puede apreciar en la torre de la izquierda que aparece en la figura 3.8) y se ha conectado al cuerpo como se puede observar en la figura 3.10. Aplicándole la simetría correspondiente al cuerpo para obtener las cuatro torres.

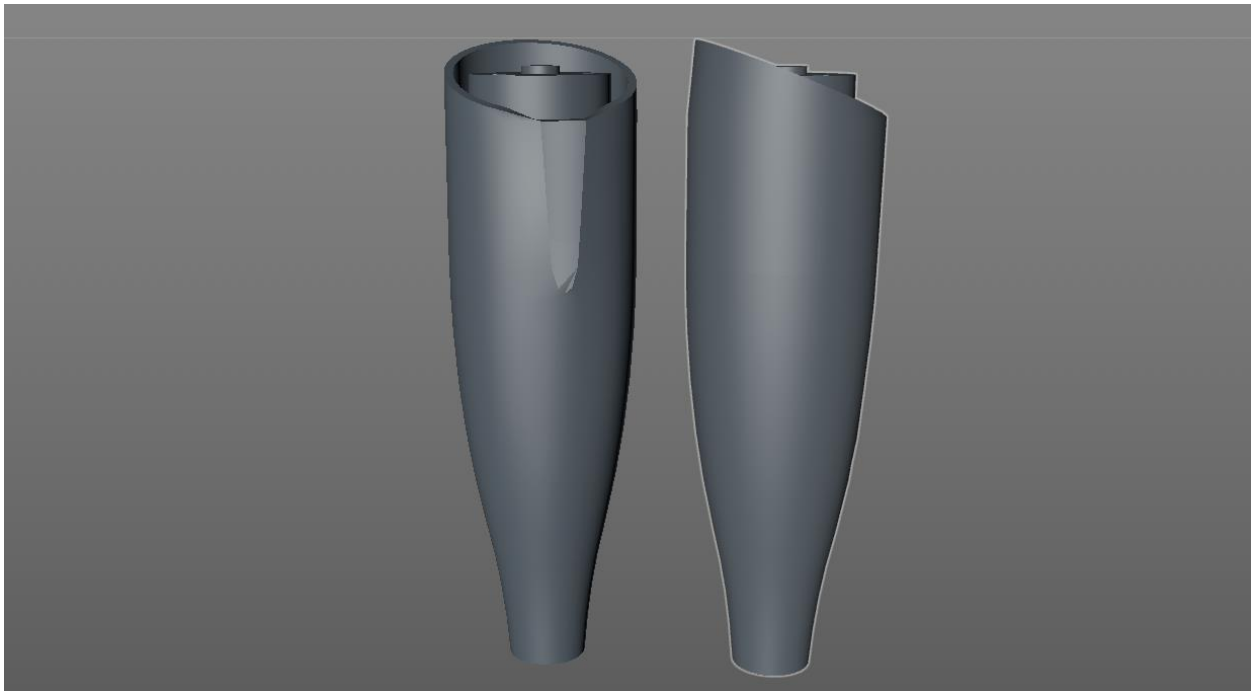

Figura 3.8 - Visión de la torreta motora desde varios puntos de vista

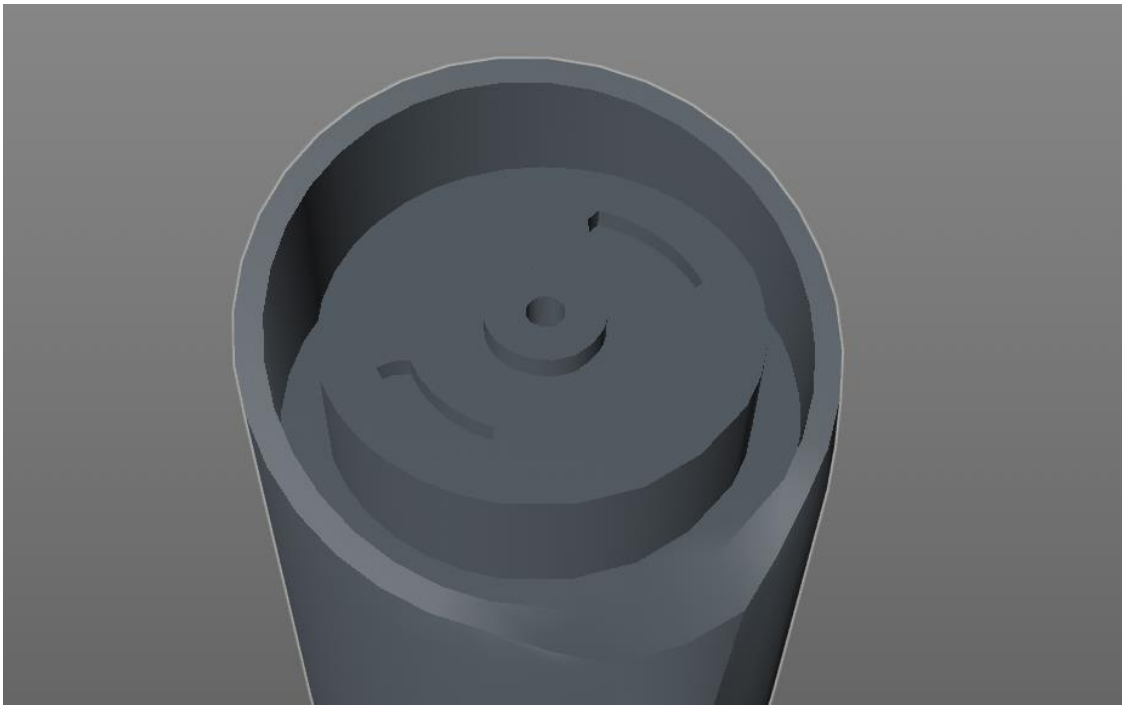

Figura 3.9 - Detalle torre (motor insertado en ella)

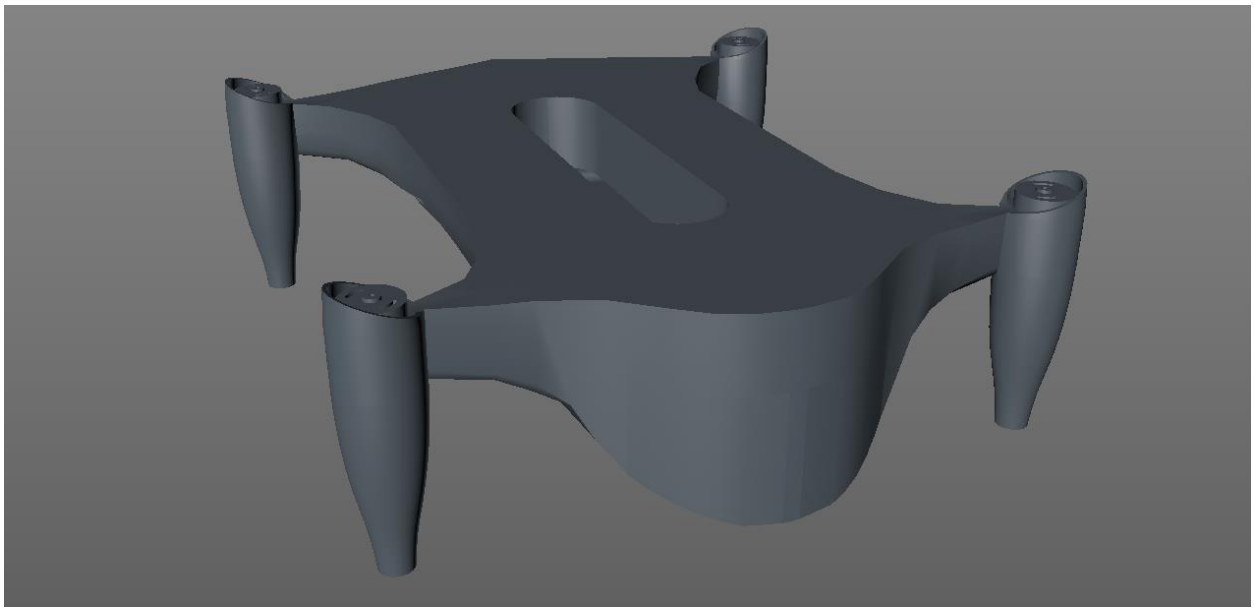

Figura 3.10 - Detalle cuerpo con torres unidas

Un detalle muy importante es el que se muestra en la figura 3.11: si se unen los centros de los huecos de cada motor después de aplicar la simetría se puede observar que el plano resultante es un cuadrado perfecto.

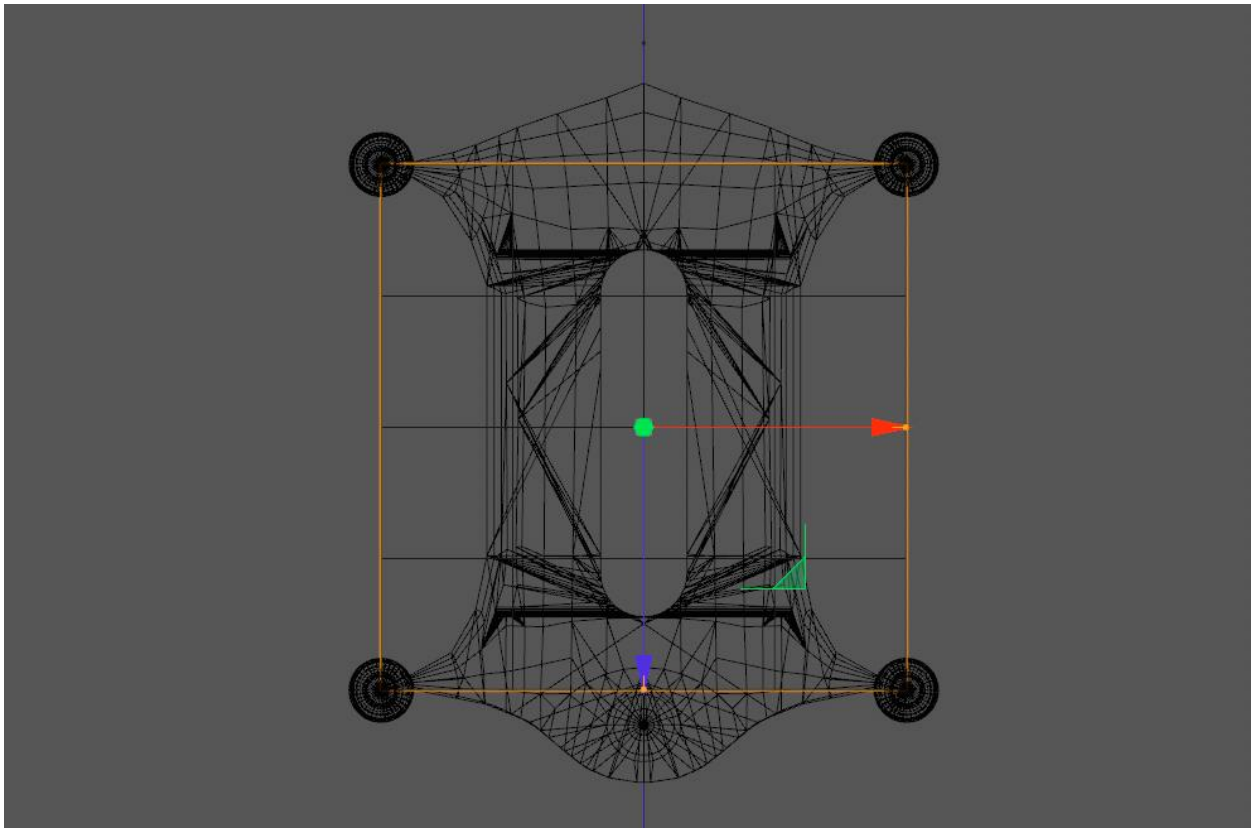

Figura 3.11 - Vista superior, detalle simetría

3.3 Diseño de la cámara

El diseño de la cámara ha estado inspirado en la reciente cámara de la marca DJI: la zenmuse X7.

Externamente los pasos seguidos para su realización han sido los siguientes:

1. El sensor es un prisma octagonal al que se le ha modificado la trasera para que no sea plana.

Confróntese la figura 3.12, que muestra una vista lateral de la cámara. Después se le ha añadido un cilindro con poca altura para poder inserter bien el objetivo.

2. En segundo lugar se ha introducido el objetivo, que es la pieza que más diseño tenga del quadrotor. El exterior del cilindro se puede observar en la figura 3.13. El interior, que se explicará a continuación, se puede mirar en la figura 3.14, que muestra la misma vista que la 3.13, pero eliminando el cristal externo del objetivo. Todo el diseño ha sido mediante subdivisions concéntricas. Así se han ido creando todos los niveles que tiene el interior del objetivo. La única superficie distinta es el cristal central, que se ha suavizado para que tenga forma de semiesfera.
3. Por último se le ha añadido a ambos lados la sujeción. Se puede mirar el resultado final en las figuras 3.12 y 3.13.

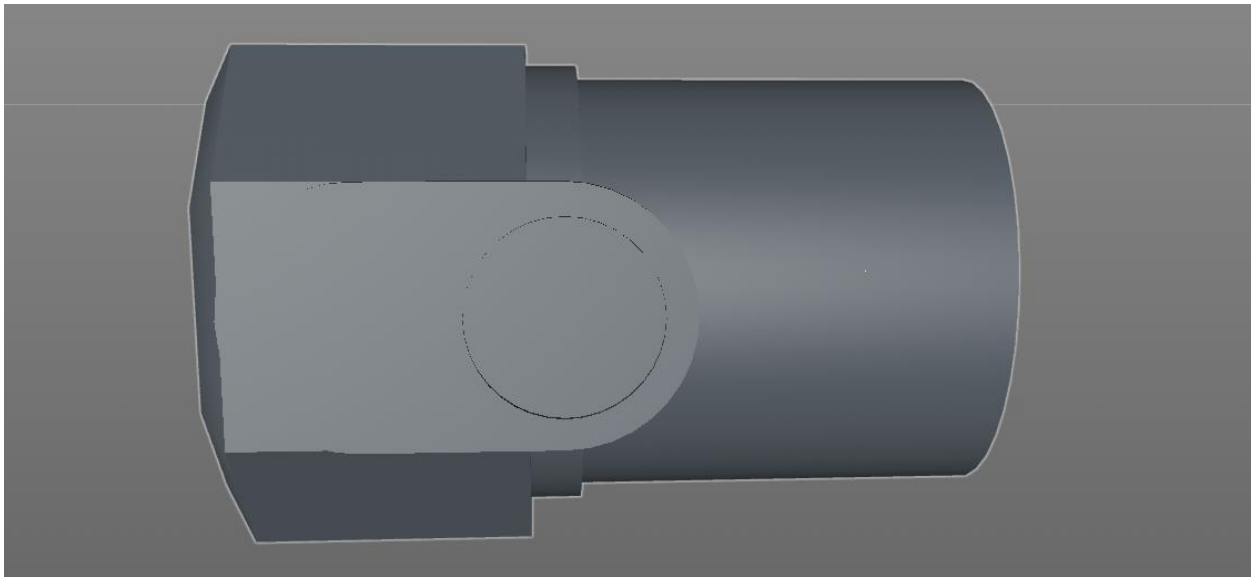

Figura 3.12 - Vista lateral de la cámara

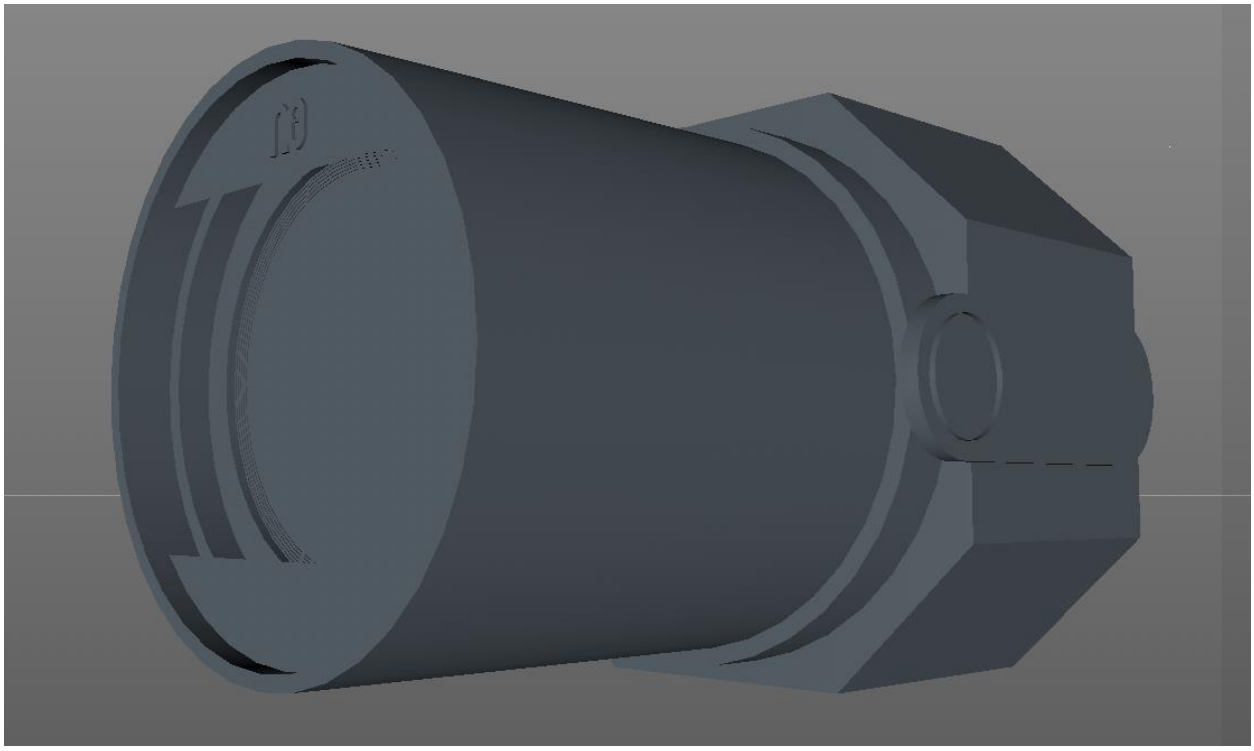

Figura 3.13 - Otra vista de la cámara

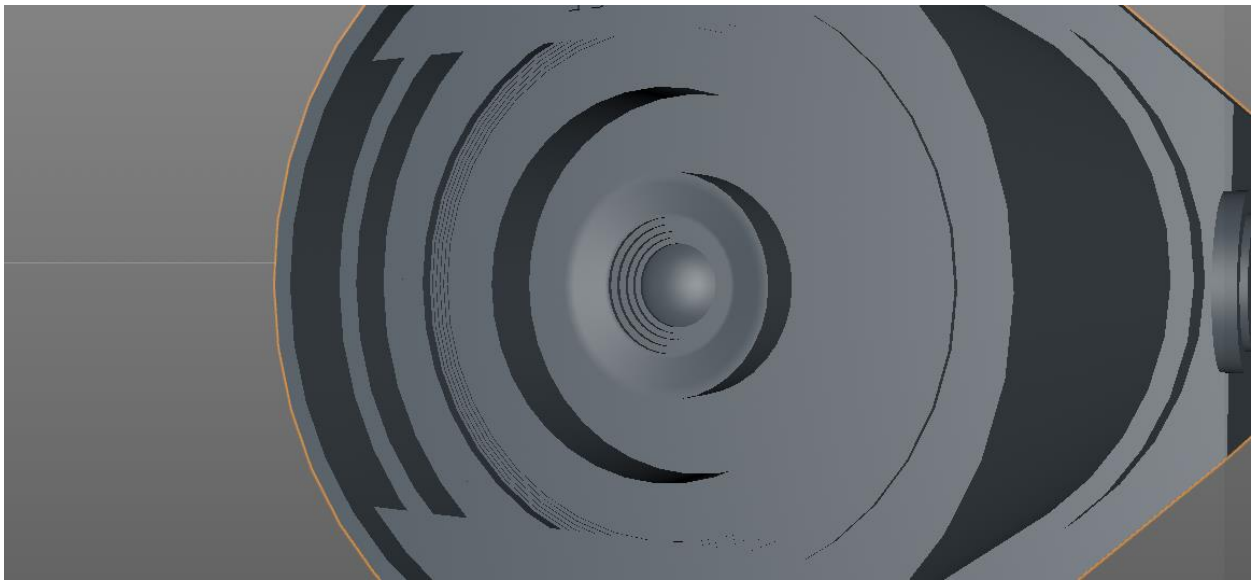

Figura 3.14 - Detalle sensor cámara (sin cristal delantero)

3.3.1 Diseño del Gimbal de 3 ejes

Un gimbal de 3 ejes consiste en un estabilizar de 3 ejes mecánico. Se realiza mediante la inserción de tres motores para controlar cada eje. Esto consigue ayudar a la cámara a estabilizar el video o mantener fijado un punto dentro de la imagen aunque el quadrotor se mueva, existiendo límites como se puede obviar.

El diseño ha consistido en:

1. Creación del estabilizador del plano horizontal y la sujeción doble al eje. Puede verse en la figura 3.15. Se ha realizado mediante la subdivisiones concéntricas de un cilindro.
2. En la figura 3.16, se puede observar la unión del primer estabilizador con el del primer plano vertical. Esto compone un estabilizador de dos ejes. El diseño se ha llevado a cabo mediante una forma similar a las unions del cuerpo con las torres motoras, es decir, allargando algunas caras de los cilindros que contienen a los motores que ejercen los pares para el estabilizado.
3. Por ultimo se ha creado el ultimo estabilizador. En la figura 3.17 se puede observar que se ha construido igual que el anterior, pero añadiendo un brazo más.
4. Por ultimo se ha añadido la cámara en el hueco principal del estabilizador de tres ejes.

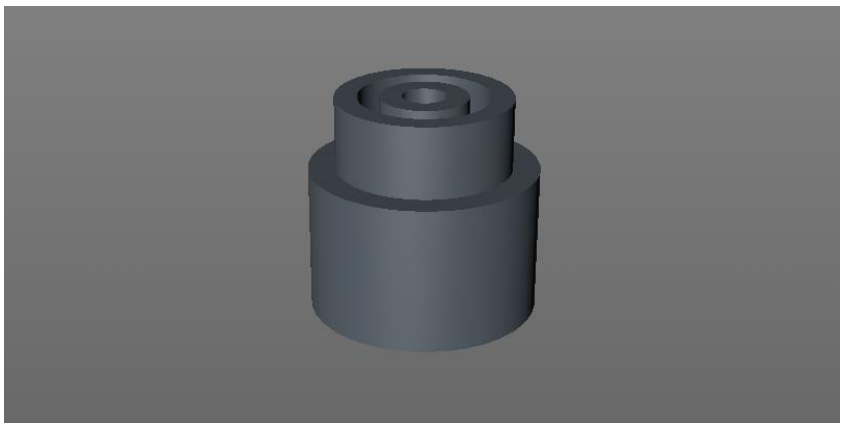

Figura 3.15 - Gimbal con estabilizador de un eje

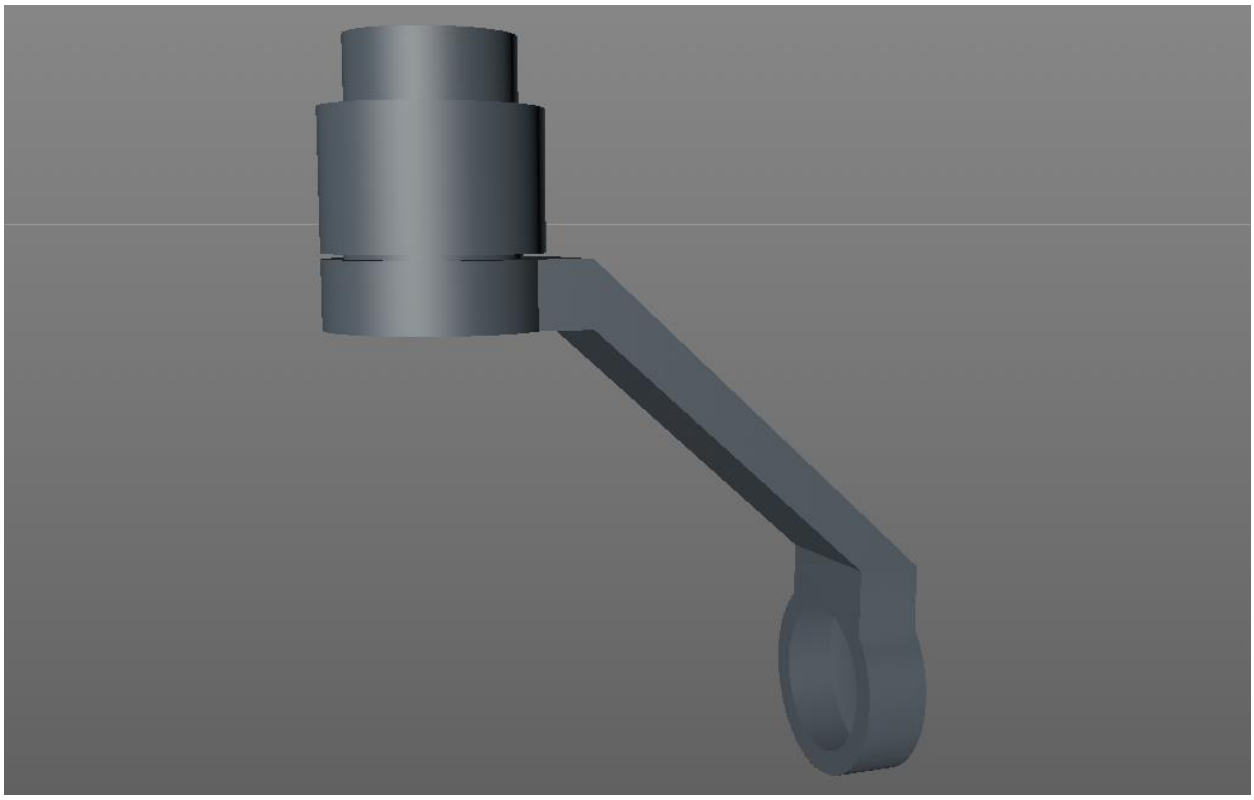

Figura 3.16 - Gimbal con estabilizador de dos ejes

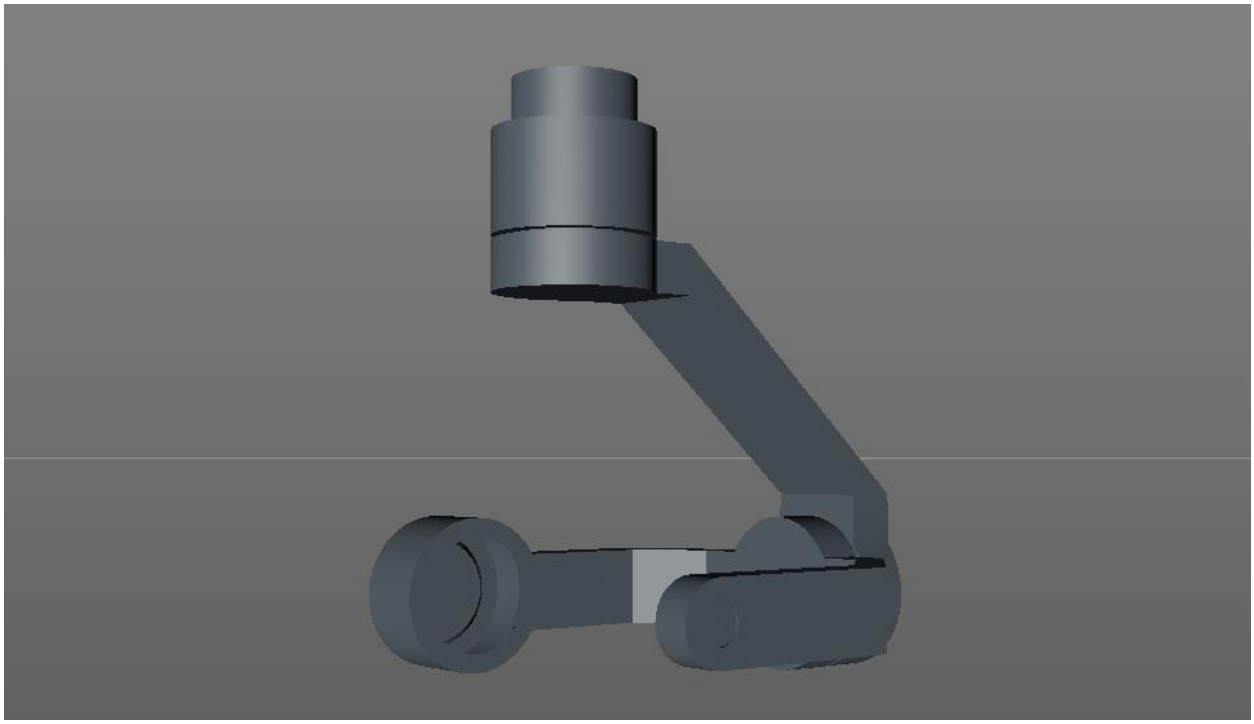

Figura 3.17 - Gimbal con estabilizador de tres ejes

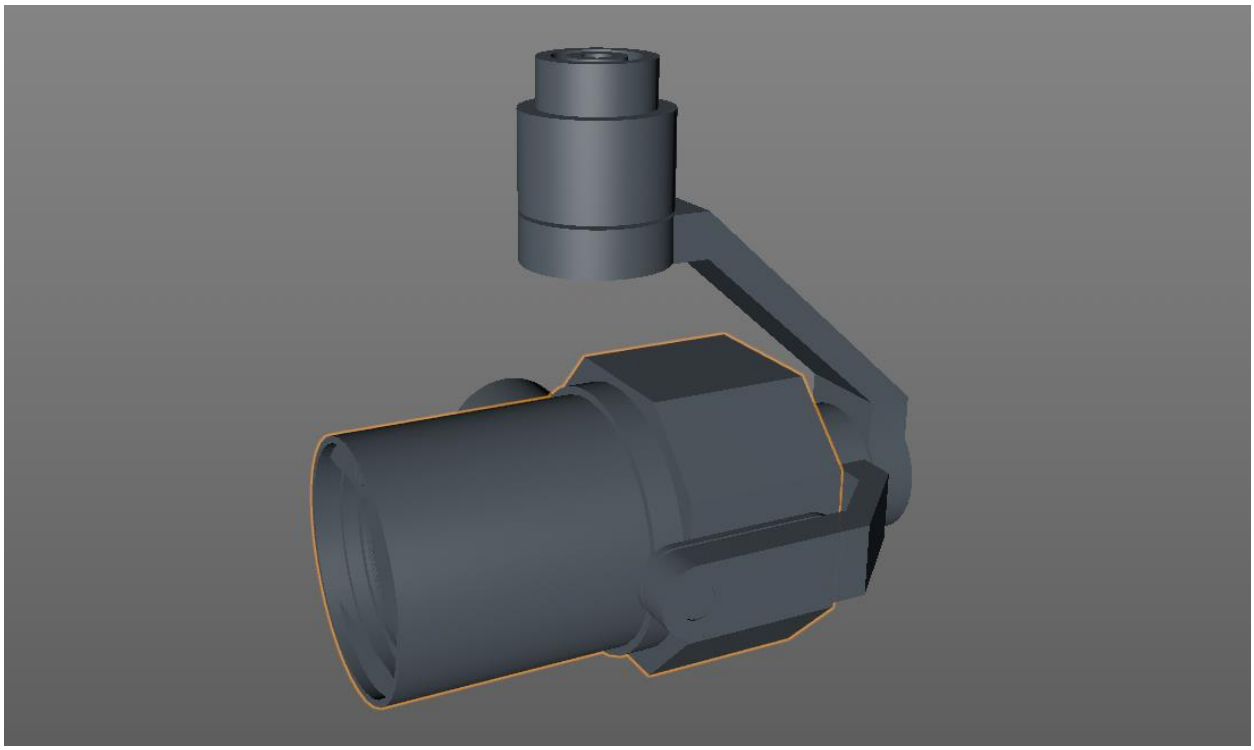

Figura 3.18 - Sistema completo

3.5 Diseño de las hélices

El diseño de las helices es probablemente el diseño que más tiempo ha llevado, devido a superficie de cada ala. Un detalle del ala se puede observar en la figura 3.19, que ha sido captada a conciencia con las subdivisiones internas que componen al ala. El ala es un plano de 7 vértices. Una vez dada la forma se le ha aplicado una transformación que aumenta todas las divisions internas. Con esto se consigue una superficie perfectamente redondeada, sin picos observables. Después se la ha dado aportado un pequeño grosor para que tenga resistencia. Antes del ala se ha diseñado la plataforma de sujeción de la hélice. Es la que se puede observar en la figura 3.21 (vista superior) y 3.20 (vista inferior). Se compone de tres elementos: el elemento situado en medio, que es donde están sujetadas las dos alas; y los dos elementos superior e inferior, que aportan la sujeción de la hélice con el eje. Esta sujeción es debida a que, mediante dos tornillos pasantes, de rosca de estrella, fijan entre sí las tres formas. Mediante simetría inversa, se ha conseguido la segunda ala y la segunda parte de la hélice.

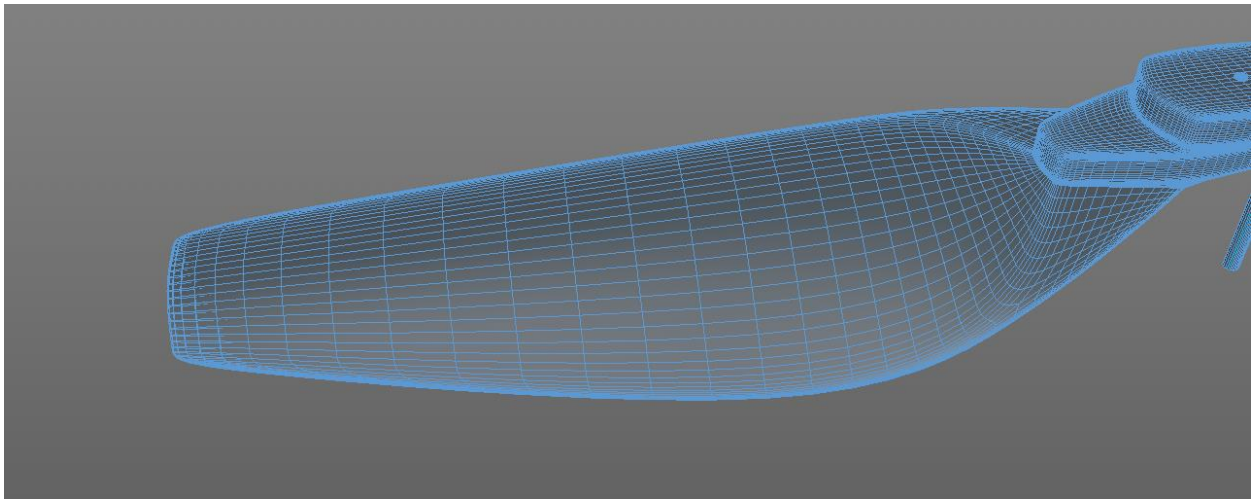

Figura 3.19 - Ala de una hélice

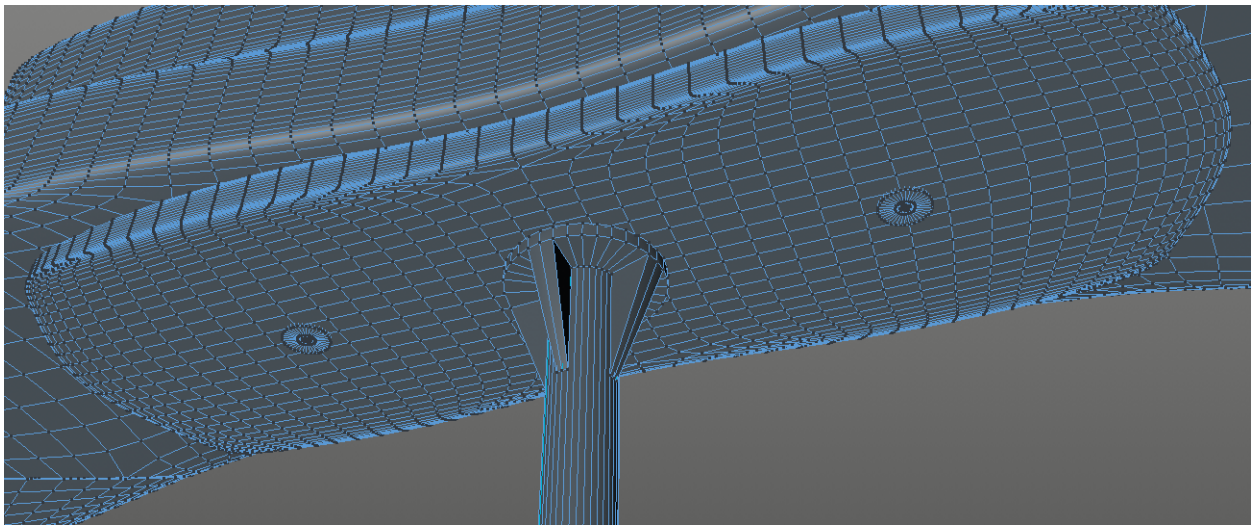

Figura 3.20 - Vista inferior hélice (pasantes y eje)

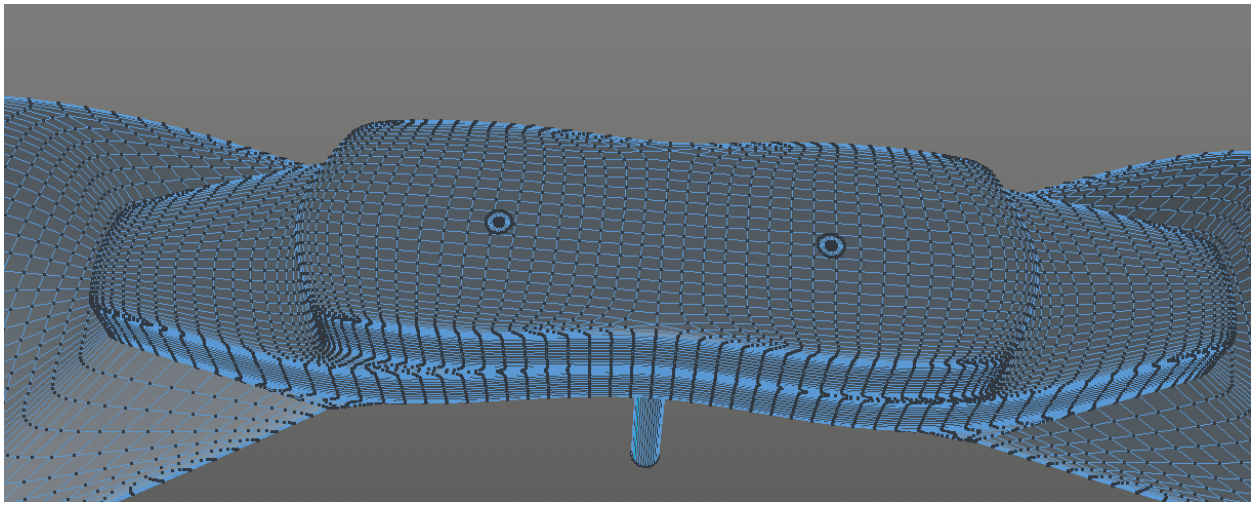

Figura 3.21 - Vista superior hélice (pasantes y sujeción)

Hay un detalle importante que es digno de mención para aclaración de dudas. El eje es parte de la hélice con el único objetivo de que en el simulador final (Unity3d) el eje también gire.

En la figura 3.22 se pueden observar las cuatro vistas de una hélice una vez que se le ha aplicado la simetría inversa respecto a su eje central. En la figura 3.23 se puede observar que, mediante simetría en los ejes X e Y, se han obtenido las otras tres helices.

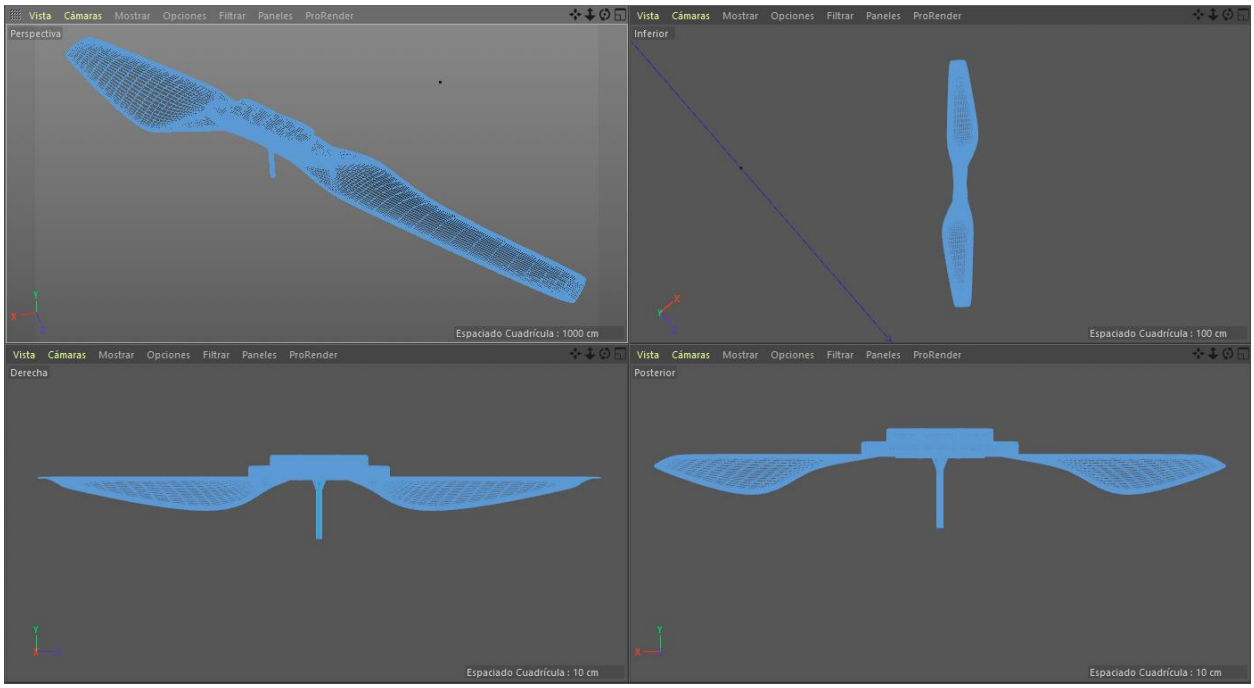

Figura 3.22 - Vistas hélice completa

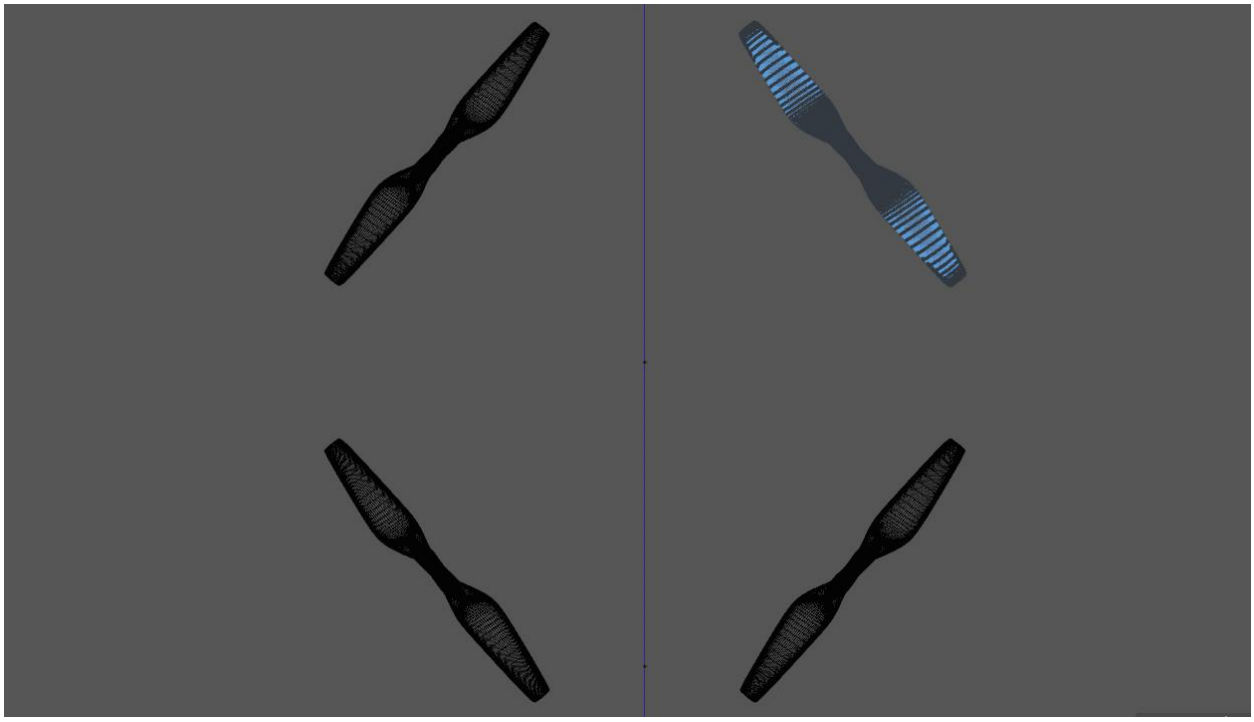

Figura 3.23 - Vistas simetrías helices

En la figura 3.24 se puede ver cómo quedó el resultado de unir las helices con el cuerpo del quadrotor.

Por ultimo se muestra en la figura 3.25 un detalle de la conexión de las helices con un motor situado en lo alto de una torre motora.

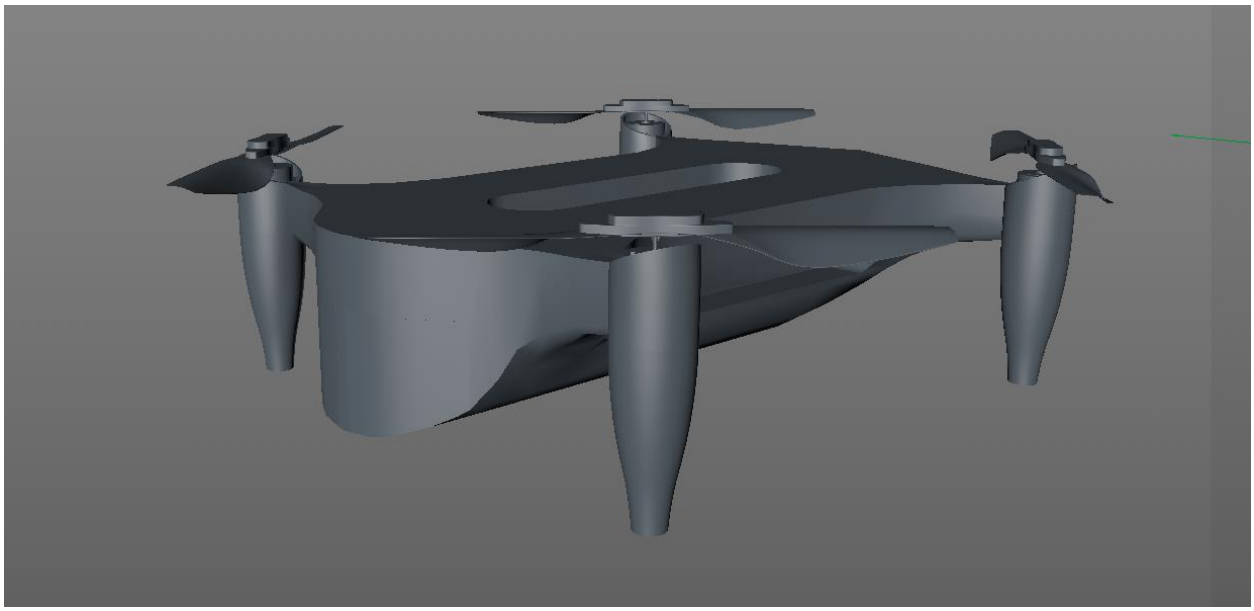

Figura 3.24 - Vista quadrotor con hélices

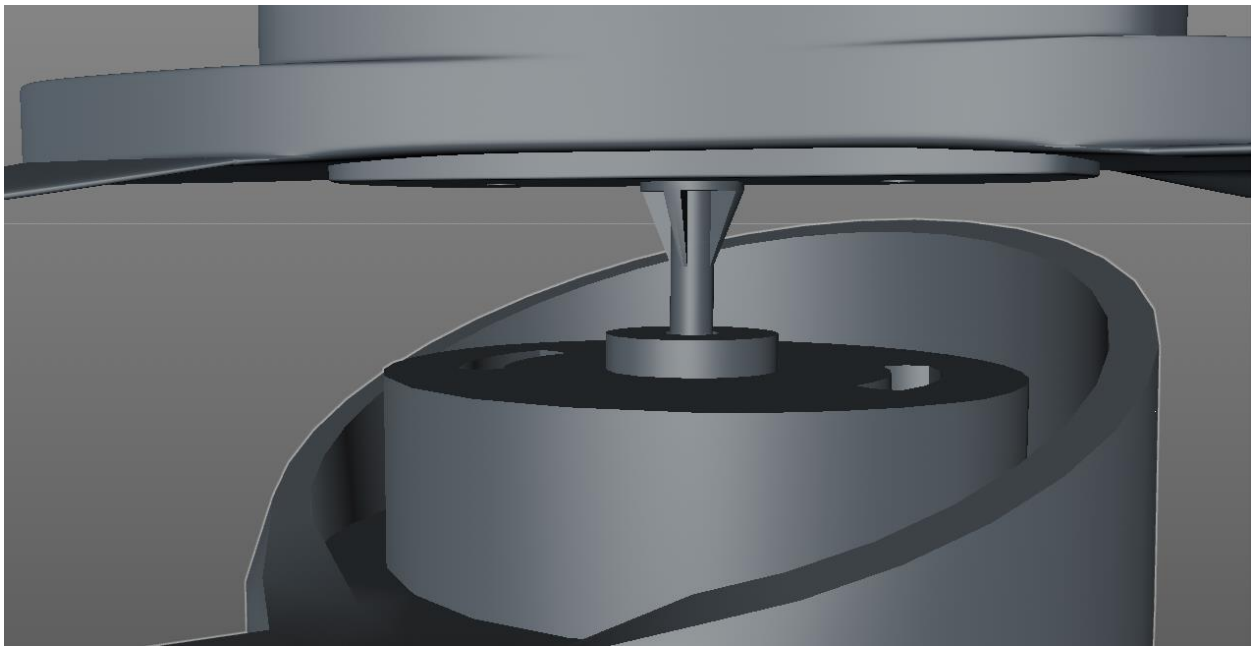

Figura 3.25 - Detalle conexión helices y motores

3.6 Personalización con una marca

Es el penúltimo paso del diseño. Se han seguido tres pasos para personalizar el quadrotor con un logo, en este caso, el logo de la startup nO:

1. Se ha pasado el logo desde un archivo vectorizado a un diseño en 3d como se muestra en la figura 3.26.
2. Se ha realizado una extracción del logo en la parte superior frontal de la superficie de arriba del cuerpo. Mírese la figura 3.27.
3. Además se ha añadido de manera que sobresalga en el último motor del gimbal de tres ejes de la cámara y en el objetivo de ésta como se puede observar en las figuras 3.28 y 3.29 respectivamente.

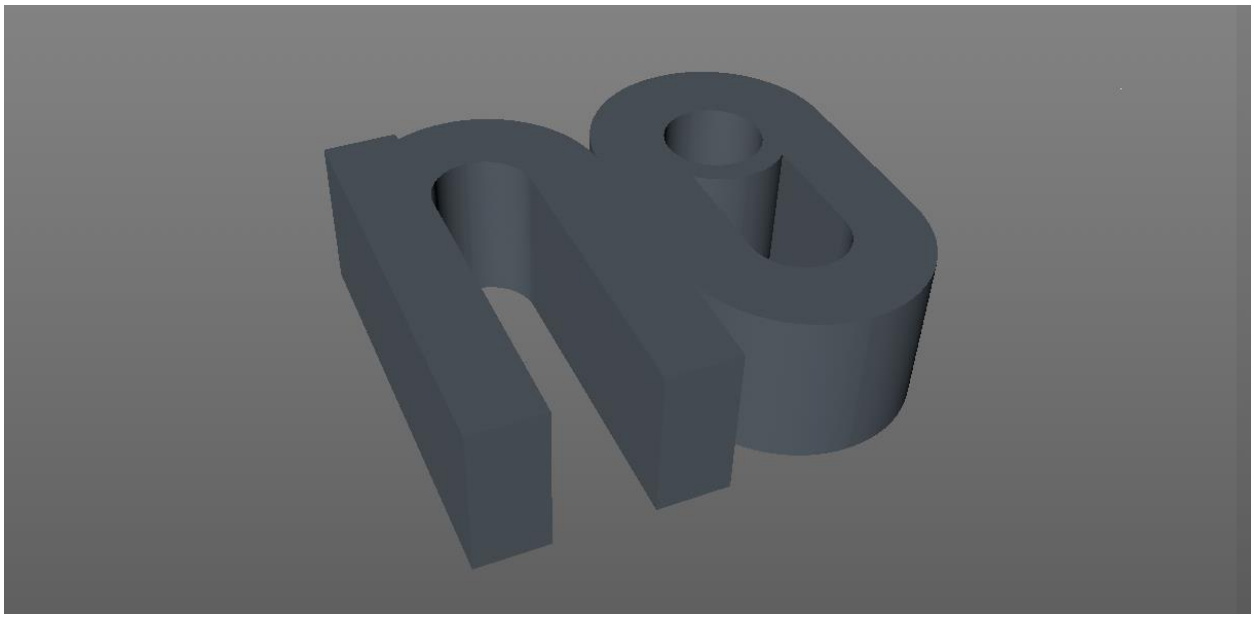

Figura 3.26 - Logo nO en 3d

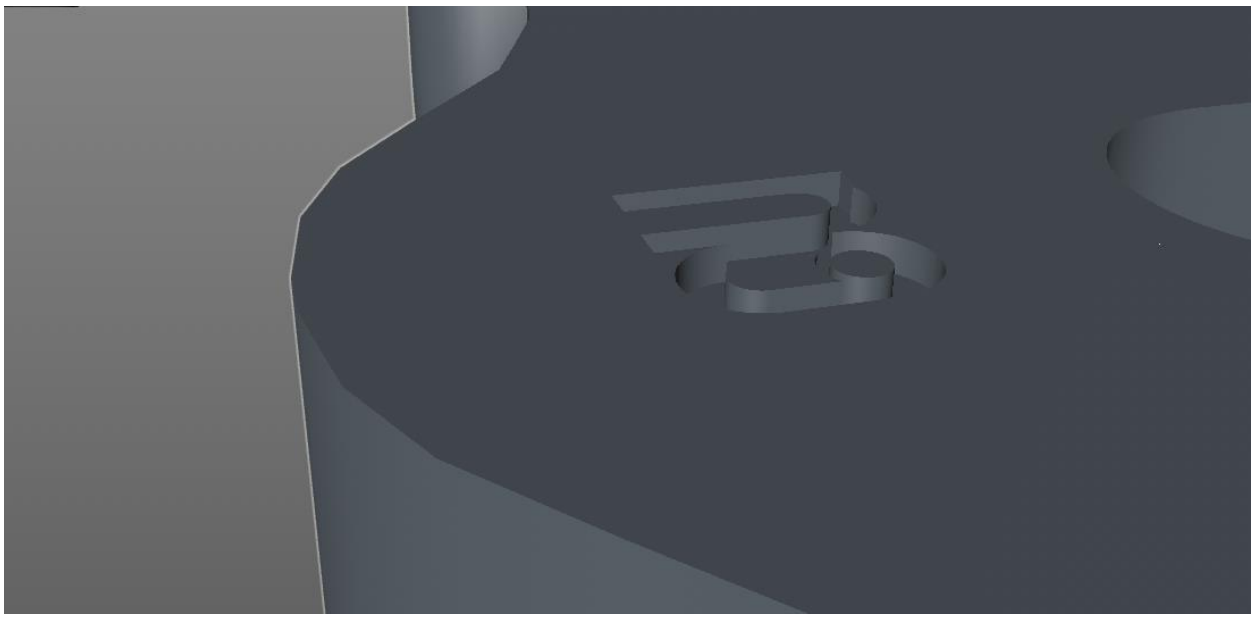

Figura 3.27 - Logo nO en 3d

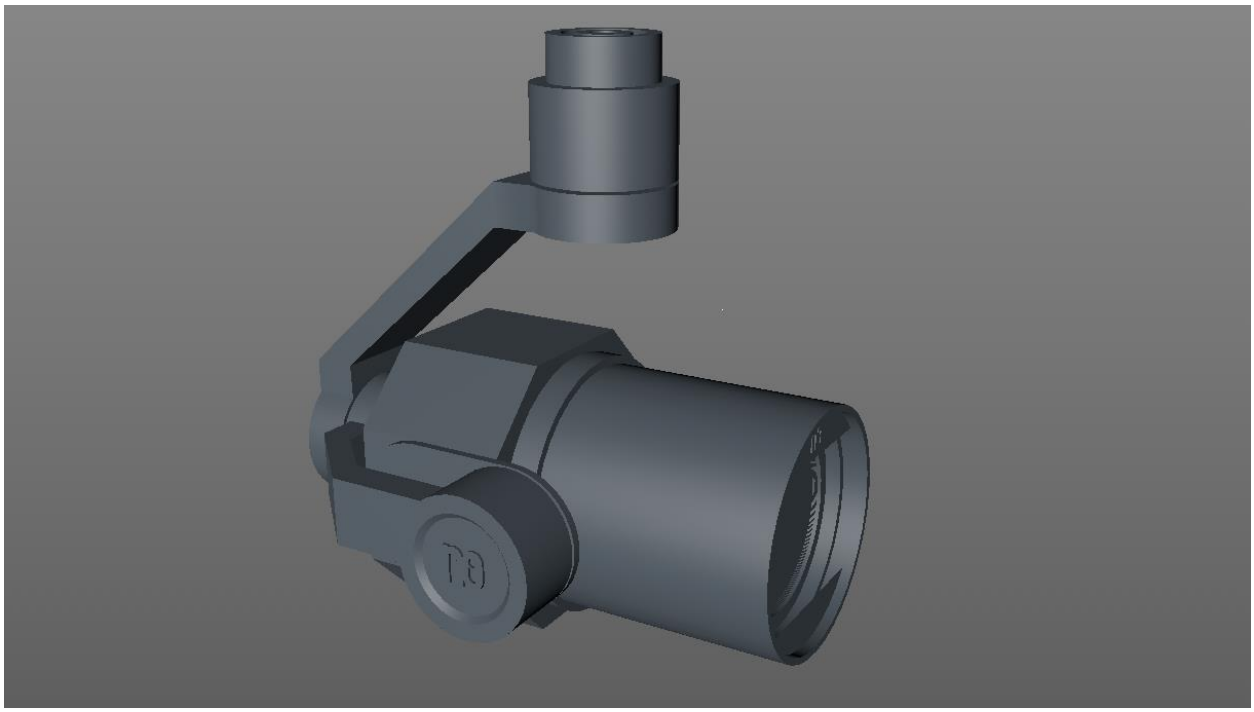

Figura 3.28 - Logo nO situado en el gimbal de 3 ejes

Figura 3.29 - Logo nO situado en la parte frontal de la cámara

3.7 Adición de texturas

Esta parte consiste en añadir las texturas necesarias a cada superficie para darle la impresión de que está realizado o construido con un tipo de material concreto.

Desde la figura 3.30 hasta la figura 3.39 muestran los materiales elegidos para cada pieza.

Al final se puede decir que el dron está realizado mediante plástico rígido, cristal, metal para zonas críticas y goma para otras zonas como la parte superior del cuerpo.

En las figuras 3.40 y 3.41 se pueden observar dos vistas del quadrotor terminado.

3.7.1 Cámara más gimbal de tres ejes

Figura 3.30 - Cámara con texturas

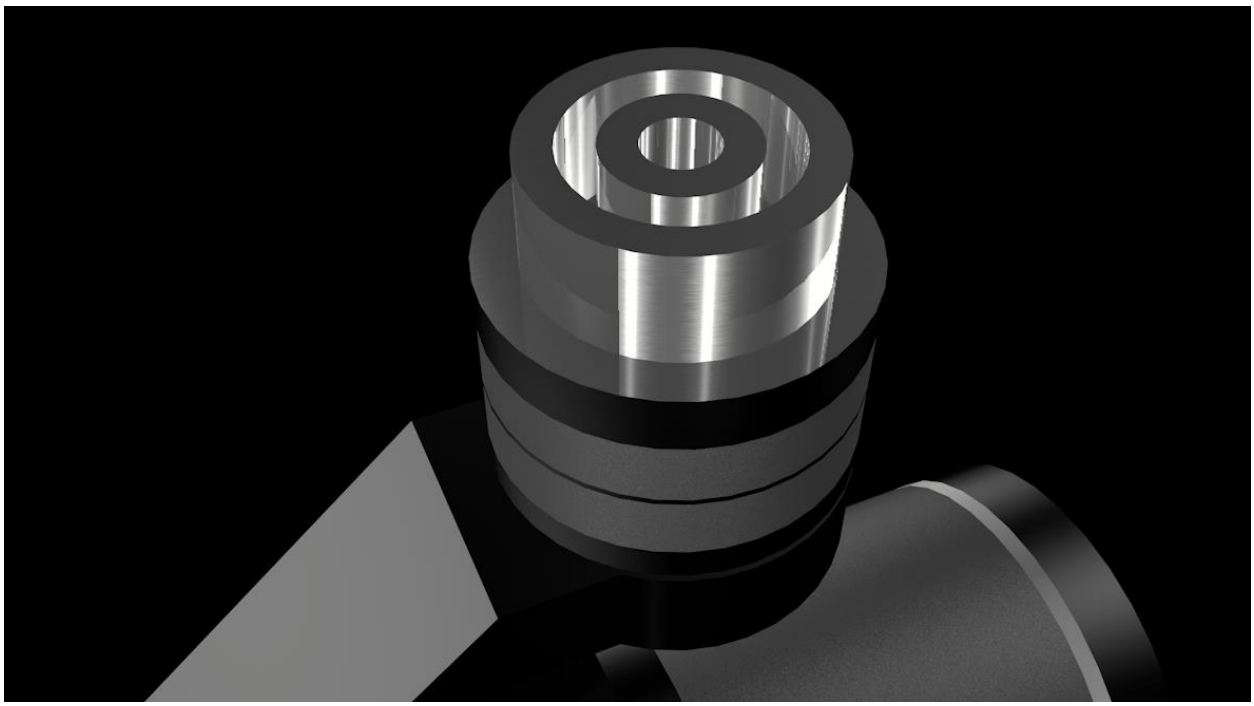

Figura 3.31 - Conexión gimbal con texturas

Figura 3.32 - Cámara con estabilizador de 3 ejes completos

3.7.2 Hélices con sus ejes

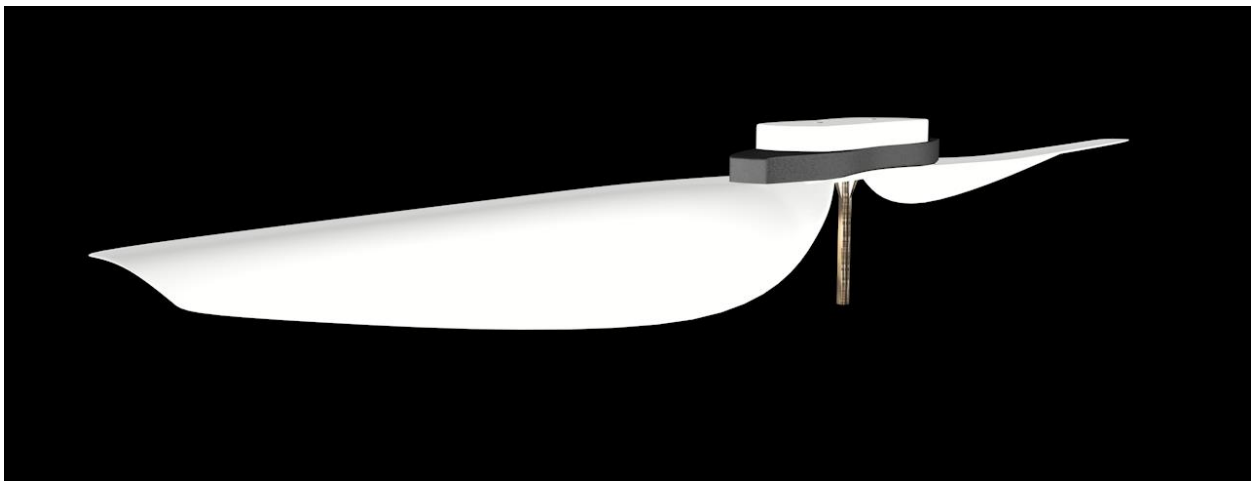

Figura 3.33 - Hélice con texturas

Figura 3.34 - Vista inferior hélice con texturas

3.7.3 Cuerpo completo

Figura 3.35 - Cuerpo del quadrotor con texturas

Figura 3.36 - Detalle anclaje cámara con textura

Figura 3.37 - Detalle cámara anclada con cristal de protección retirado

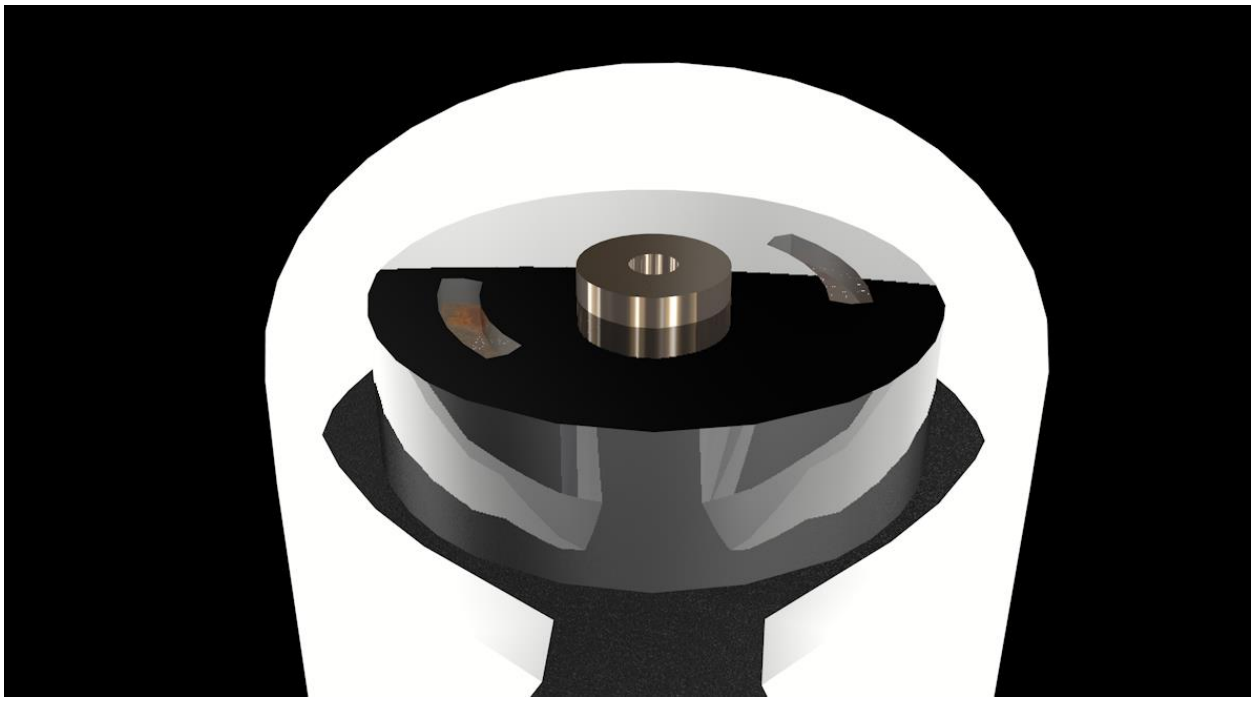

Figura 3.38 - Detalle motor con texturas

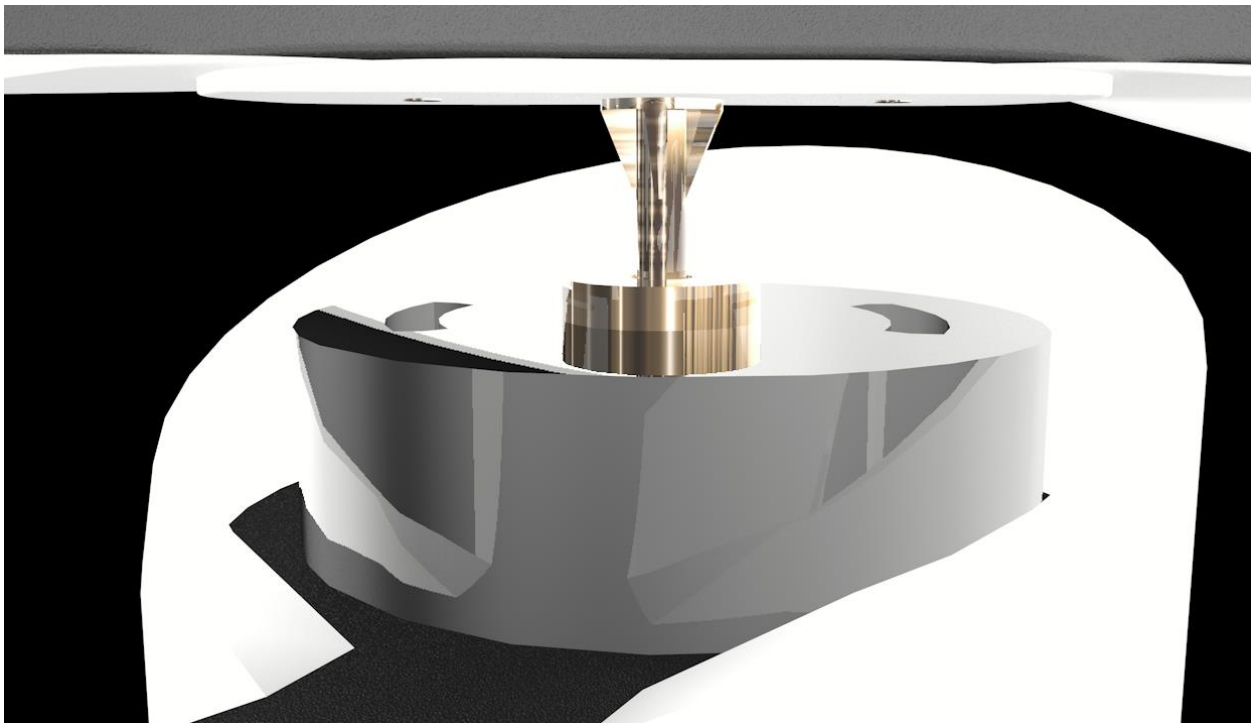

Figura 3.39 - Detalle motor con hélice

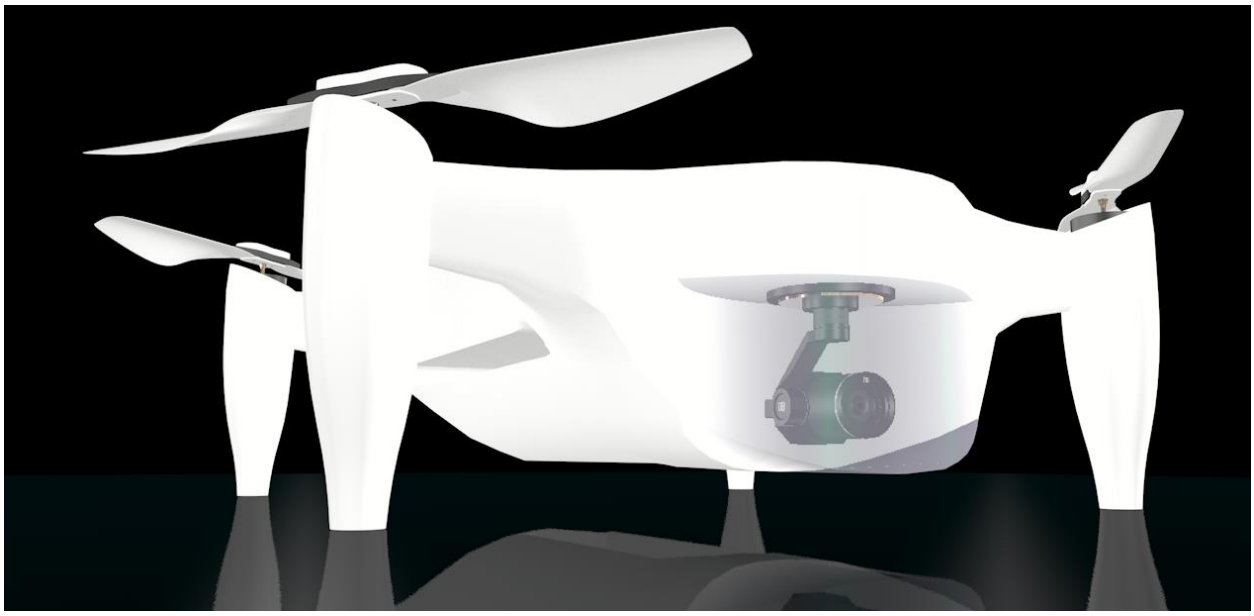

Figura 3.40 - Vista diagonal quadrotor finalizado

Figura 3.41 - Vista superior del quadrotor terminado con detalle

3.8 Preparación para Unity3d

Se ha tenido que simplificar para que reconociese las texturas y centrar los ejes.

Una vez terminado el diseño del quadrotor, se ha exportado al formato .fbx para que lo reconociese el entorno de Unity3d.

Se ha introducido en Unity3d y se ha comprobado que las texturas no es capaz de reconocerlas. Es uno de las dificultades más grandes de estos entornos: los programas de diseño hoy en día permiten alcanzar un nivel de detalle que son inalcanzables por los prorgamas de virtualización como pueden ser el Unity3d o el Unreal Engine. Está a punto de salir una nueva version de Unreal Engine (UE5) que parece que podrá solucionar este problema, entre otras cosas.

Otro problema que fue detetctado en Unity3d fue que los giros se producen según los ejes creados en Cinema4D.

El último problema detectado fue el de las normales de las superficies. Aquella superficie que su normal no fuera externa (apuntase hacia fuera del objeto), era transparente en Unity3d.

Al final se solucionaron hacio estos tres cambios:

1. Reduciendo a cuatro texturas todas las existentes.
2. Ajustando los ejes de todos los objetos que queríamos que girasen como las hélices, el gimbal y la cámara.
3. Cambiando todas las normales para que fuesen externas.

A modo de ilustración, en la figura 3.42, se pueden apreciar estas tres correcciones.

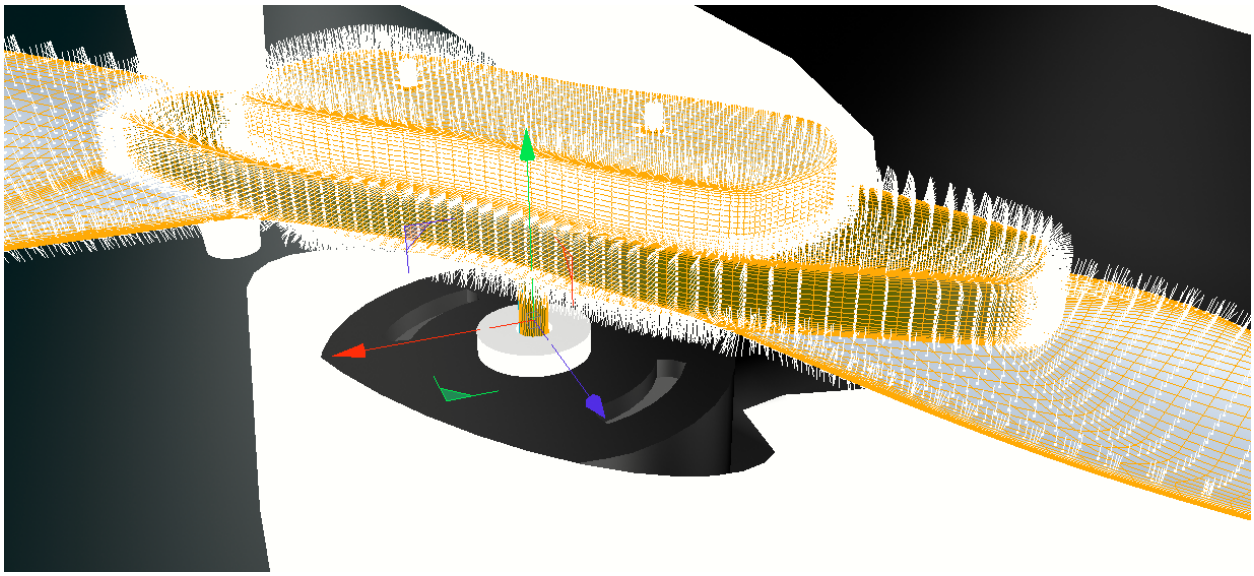

Figura 3.42 - Detalle de ejes centrados, normales externas y reducción de texturas

4 CREACIÓN DEL SIMULADOR FINAL

Si lo puedes soñar lo puedes hacer.

Walt Disney

En este último capítulo del proyecto se trata de convertir el modelo dinámico creado en 3d en un sólido rígido real, para después poder controlarlo, mediante los controladores diseñados en la primera parte del proyecto. Para ello utilizaremos un entorno virtual llamado Unity3d [7].

Unity3d es una herramienta que permite crear aplicaciones multiplataforma, en 2d como en 3d. Entre otras herramientas internas destacables de Unity3d es su motor de física. Este motor permite simular la realidad física tal cual es, o al menos con mucha semejanza.

A continuación, se irá desarrollando todos los pasos seguidos para montar el simulador final. Este simulador busca como gran objetivo ser un simulador versátil, abierto y expandible. Todas estas características son las que le dan valor al simulador, y no que el diseño sea más estético o no, ni que el diseño de los controladores sean mejores o peores, que de hecho no se ha buscado el mejor controlador. Pero sí se ha buscado que la inserción del mejor controlador ya diseñado se pueda realizar sin dudas ninguna en un instante.

4.1 Inserción del diseño en 3D

El primer paso realizado ha sido insertar el diseño en 3D visto en el capítulo anterior. Es importante recordar el problema visto al final del anterior capítulo sobre la no detección de texturas/materiales, el necesario sentido externo de las normales de todas las superficies y la centralidad de los ejes de rotación para el buen funcionamiento del simulador.

Para insertar el modelo ha sido tan sencillo como añadir en la carpeta de assets el modelo exportado en formato *.fbx*. Unity3d reconoce automáticamente el modelo y lo inserta dentro de la barra del Proyecto, como se puede observar en la figura 4.1.

En la figura 4.1 también se puede observar como detecta todos los elementos que lo componen. Esto es importante a la hora de tratar por separado las hélices, el cuerpo y cada componente de la cámara (Gimbal x3 componentes).

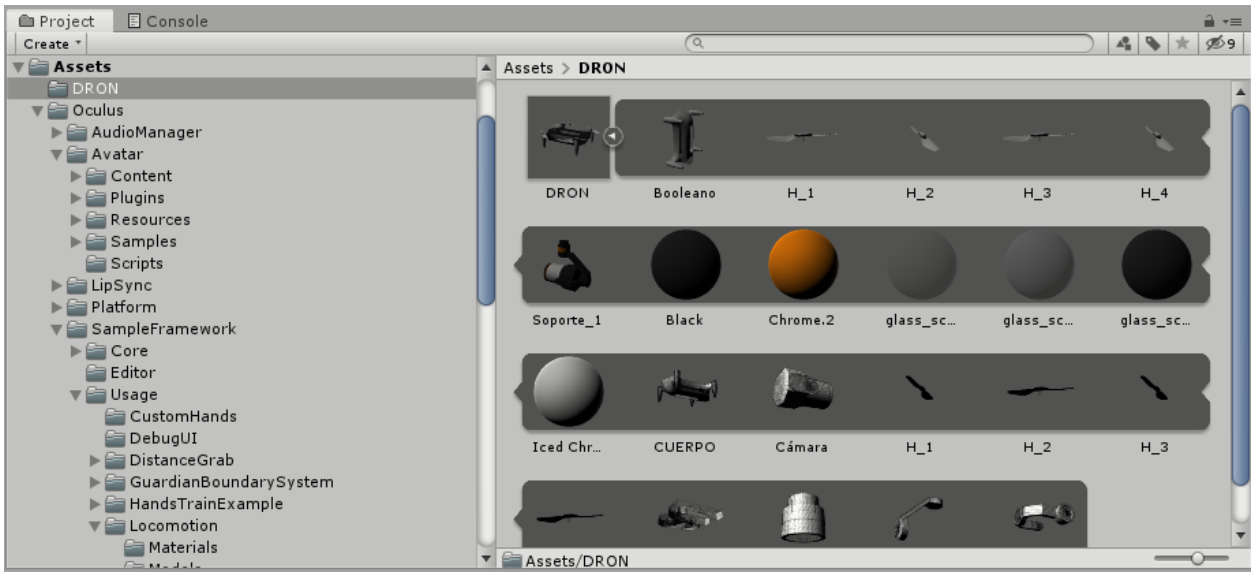

Figura 4.1 - Barra de assets de Unity3d

Como se ha resuelto en el capítulo anterior, no se explicará más el tema, salvo el amtiz de las texturas. Si se comparan las figuras 4.2 (diseño simplificado en el entorno de Cinema4D) y 4.3 (diseño simplificado insertado en Unity3d) se puede observar que varias tecturas han cambiado.

En primer lugar hay que nombrar que las texturas se redujeron a 4: negro mate para todas los materiales cercanos a este color como pueden ser la goma, el plástico, etc. Blanco para todas aquellos materiales que ya eran de este color (se han añadido algunos como la goma de la cámara para aumentar el zoom o el enfoque, antes era de goma negra); cristal transparente para todos aquellos que eran cristales (el protector de la zona de la cámara más los dos de la cámara: el interno y el externo). Por último un beige para los materiales que se querían resaltar. Éste último es donde se puede observar el mayor cambio: en Cinema4D se asimila a un beige, mientras que en Unity3d es un naranja intenso. Este cambio se puede observar por ejemplo en la cámara y en el interior de los motores.

Figura 4.2 - Modelo simplificado en Cinema4D

Figura 4.3 - Modelo simplificado en Unity3d

Una vez insertado el modelo, se ha creado otro objeto 3d con forma de prisma rectangular, como se aprecia en la figura 4.4. Éste objeto se le ha llamado *DroneBody*. Éste objeto es el que instancia los programas que se explicarán más adelante. Esto implica que las fuerzas se aplicarán a este prisma, y, por filiación, a todos sus hijos. La única propiedad que se le ha asignado a este objeto, aparte del programa instanciado, es la de ser un sólido rígido. Esto nos permite utilizar con este objeto el motor físico de Unity3d. Finalmente se le indica que en la imagen no aparezca y resulta como la figura 4.3.

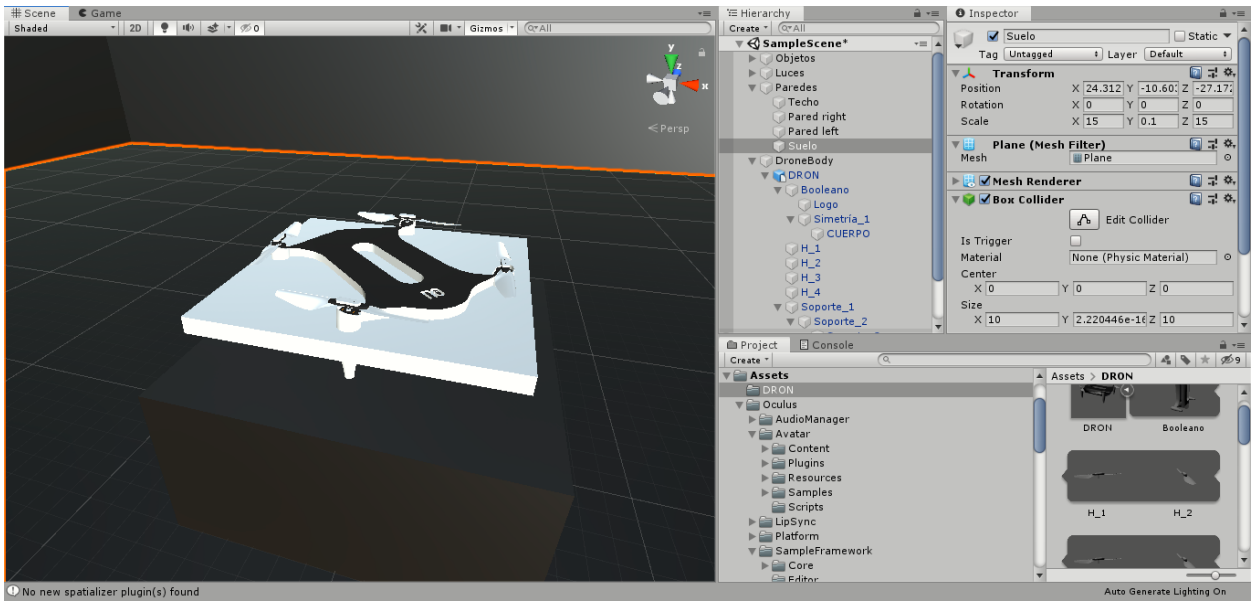

Figura 4.4 - Objeto prismático controlable

A este objeto se le ha asignado como hijo el quadrotor diseñado en cinema4d, como se puede apreciar en la figura 4.4, con el nombre de *DRON*. Se ha posicionado exactamente en el centro, con un cierto ángulo en yaw para conseguir que el sitio donde se apliquen las fuerzas sea finalmente el centro de las helices. Una vez todo listo, se configuró los *colliders* que son aquellas superficies que rodean a los objetos para saber si han tenido contacto o no con otros objeto. Se pueden utilizer para muchos usos, pero el principal es para que el motor físico detecte colisiones. En la figura 4.5 se muestra el *collider* del quadrotor. Todos los demás objetos tienen

también colliders, si no, se atravesarían como si no existiesen.

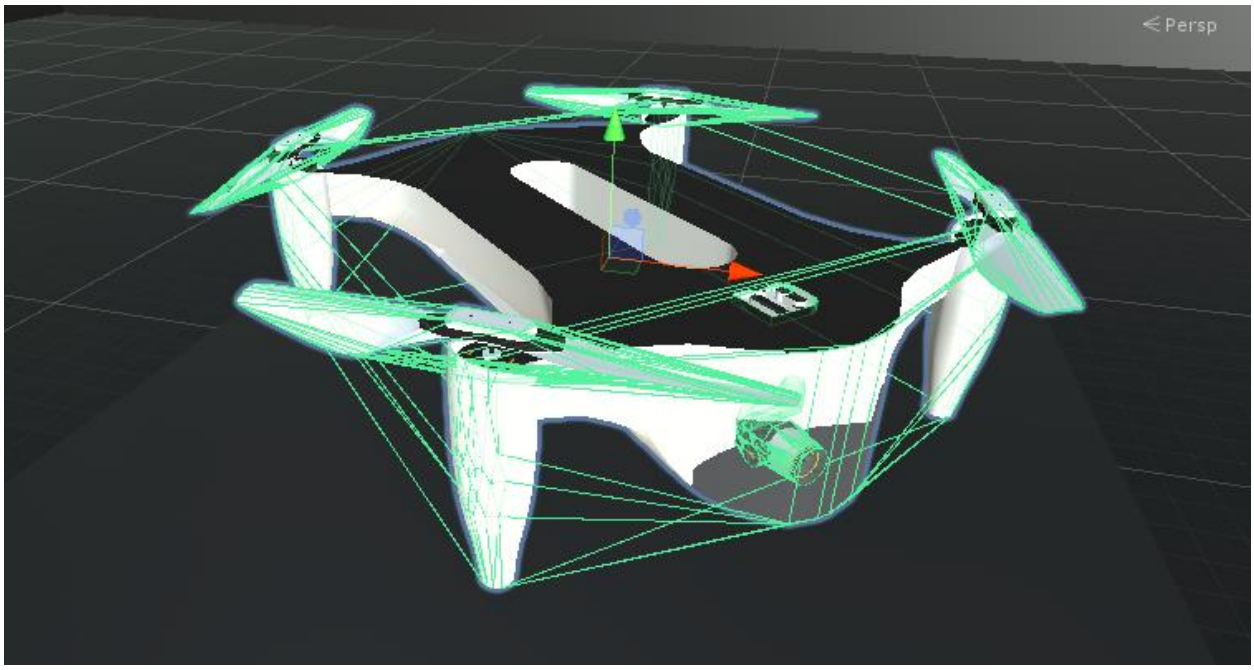

Figura 4.5 - Collider del quadrotor

Debido a una actualización del entorno Unity3d, no se permite introducir un *mesh collider* a sólidos rígidos, es decir, un collider perfecto que se ajusta totalmente al diseño en 3d. Como solución está un collider hecho con triángulos que se ajusta en cierto sentido bastante bien. En la figura 4.5 se puede comprobar estos triángulos.

A continuación se diseñó toda la arquitectura de programación para dar ese aspecto modular ampliable al simulador.

El siguiente paso ha sido montar un modelo para el quadrotor. Y posteriormente, migrar todos los códigos diseñados en Matlab a este entorno.

4.2 Diseño de una arquitectura expandible y modular de programación

La gran capacidad de este simulador es su arquitectura expandible y modular que contiene la programción con la que está hecho.

En este capítulo se hablará sobre todo del aspecto ampliable. Este aspecto es dado por el orden utilizado al programarlo. Los programas creados en Unity3d han sido los siguientes:

- DIRECTIVAS.cs.
- MacroQuad.cs.
- QuadModel.cs.
- QuadControl.cs.
- QuadPlan.cs.

Estos cinco programas implementan todo lo necesario para que un quadrotor pueda funcionar autónomamente con un nivel alto de funcionamiento. El único programa que se instancia es el MacroQuad. Este programa es el que gobierna a alto nivel todos los demás programas, guardando todas las variables estáticas de los demás programas. Esto permite que creando otro programa MacroQuad para otro quadrotor, podrían funcionar independientemente los dos quadrotors sin ninguna conexión. Esto es debido a que todos los demás programas son sin memoria. Para hacer esto es necesario crear una comunicación compleja, pero fácil de utilizar. Para esto se crearon dos programas: DIRECTIVAS.cs y MacroQuad.cs.

4.2.1 Programa DIRECTIVAS

El programa fundamental para dar orden a todas las demás implementaciones, e implanta toda el diseño de comunicación es el programa *DIRECTIVAS.cs.*

Este programa se irá explicando a continuación, y es la clave para entender todos los demás programas.

Este programa se compone de cuatro clases:

1. CONVERT: esta clase contiene todas las funciones o constantes necesarias para hacer conversiones. Nos encontramos en primer lugar dos constantes que son muy útiles para convertir de ángulos a radianes y viceversa. A continuación nos encontramos una función muy importante. Esta función aplica una transformación del Sistema de referencias utilizado en Unity3d al de Matlab, que ha sido el utilizado internamente en los controladores. Esta transformación invierte todos los ángulos y cambio los ejes Y por Z, dado que en Unity3d el eje vertical es el Y. Al final de esta transformación se realiza una correción de los ángulos para evitar problemas de saturación. Lo restante en esta clase son la creación de un tipo de dato *Vector6* que guarda los seis grados de libertad de la posición, una función que devuelve este mismo tipo de dato incializado con valores nulos, y dos funciones más que tranforman de *Vector6* a *Vector4*, que es un tipo de dato propio de Unity3d y se ha utilizado en gran medida en este Proyecto. El código de esta clase se adjunta a continuación:

class CONVERT : MonoBehaviour

```
{
    // Conversionas de ángulos
    public const double G2R = Math.PI / 180;
    public const double R2G = 180 / Math.PI;

    // Transformación de los sistemas de coordenadas de Unity al
interno. Además se incluye la corrección de los ángulos
    public static Vector6 TF (Rigidbody Quadrotor, Vector6 Ref)
    {
        // Definición de variables
        Vector6 Pos;

        // Tranformación de las coordenadas
        Pos.X = Quadrotor.transform.position.x;
        Pos.Y = Quadrotor.transform.position.z;
        Pos.Z = Quadrotor.transform.position.y;
        Pos.Roll                =                Convert.ToSingle(-
Quadrotor.rotation.eulerAngles[0] * G2R);
        Pos.Pitch                =                Convert.ToSingle(-
Quadrotor.rotation.eulerAngles[2] * G2R);
        Pos.Yaw = Convert.ToSingle(-Quadrotor.rotation.eulerAngles[1]
* G2R);

        // Corrección de los ángulos
        if  (Math.Abs(Ref.Roll  -  Pos.Roll)  >  Math.Abs(Ref.Roll  -
(Pos.Roll + 360 * G2R))) Pos.Roll += Convert.ToSingle(360 * G2R);
        else if (Math.Abs(Ref.Roll - Pos.Roll) > Math.Abs(Ref.Roll -
(Pos.Roll - 360 * G2R))) Pos.Roll -= Convert.ToSingle(360 * G2R);
        if  (Math.Abs(Ref.Pitch  -  Pos.Pitch)  >  Math.Abs(Ref.Pitch  -
(Pos.Pitch + 360 * G2R))) Pos.Pitch += Convert.ToSingle(360 * G2R);
        else if (Math.Abs(Ref.Pitch - Pos.Pitch) > Math.Abs(Ref.Pitch
- (Pos.Pitch - 360 * G2R))) Pos.Pitch -= Convert.ToSingle(360 * G2R);
        if (Math.Abs(Ref.Yaw - Pos.Yaw) > Math.Abs(Ref.Yaw - (Pos.Yaw
+ 360 * G2R))) Pos.Yaw += Convert.ToSingle(360 * G2R);
        else  if  (Math.Abs(Ref.Yaw  -  Pos.Yaw)  >  Math.Abs(Ref.Yaw  -
(Pos.Yaw - 360 * G2R))) Pos.Yaw -= Convert.ToSingle(360 * G2R);
```

```
            // Devolvemos la posición
            return Pos;
        }

        // Estructura especial para definir todos las variables que
definen un una posición en tres dimensiones
        public struct Vector6
        {
            public float X;
            public float Y;
            public float Z;
            public float Roll;
            public float Pitch;
            public float Yaw;
        }

        // Función que devuelve un Vector6 nulo
        public static Vector6 NullVector6()
        {
            Vector6 NULO;
            NULO.X = 0;
            NULO.Y = 0;
            NULO.Z = 0;
            NULO.Roll = 0;
            NULO.Pitch = 0;
            NULO.Yaw = 0;
            return NULO;
        }

        // Función que convierte un Vector4 en Vector6
        public static Vector6 V42V6(Vector4 In)
        {
            Vector6 Out;
            Out.X = In.x;
            Out.Y = In.y;
            Out.Z = In.z;
            Out.Roll = 0;
            Out.Pitch = 0;
            Out.Yaw = In.w;
            return Out;
        }

        // Función que convierte un Vector6 en Vector4
        public static Vector4 V62V4(Vector6 In)
        {
            Vector4 Out;
            Out.x = In.X;
            Out.y = In.Y;
            Out.z = In.Z;
            Out.w = In.Yaw;
            return Out;
        }
    }
```

2. DIRECTIVAS_MODEL: esta clase contiene todo lo necesario para conseguir que el modelo del quadrotor programado sea sin memoria. En primer lugar, nos encontramos dos estructuras que

contienen todos los parámetros del quadrotor: la primera, *PARAM*, contiene todos los parámetros dinámicos del quadrotor, explicados en la segunda sección de este documento, en el capítulo 2.9; la segunda estructura, *CONS*, contiene todas las constantes que deben saber los controladores para el buen control del modelo: las velocidades máxima y mínima de los motores, velocidad necesaria de los cuatro motores para que el modelo se mantenga con aceleración nula (contrarreste la fuerza), el par resultante de transformar esta velocidad de equilibrio a par virtual con el control de inversión (cfr. capítulo 2.3); y por último las constantes del modelo dinámico de orden dos diseñado (se hablará en el siguiente capítulo). Al final nos encontramos las estructuras que realizan toda la comunicación. A las estructuras que se utilizan para la comunicación se han denominado Packs. En concreto, en el pack para el modelo podemos encontrar la referencia al sólido rígido y una instancia de otra estructura que contiene todas las variables estáticas del modelo: las velocidades de referencia y la velocidad de los motores un instante pasado y dos intantes pasados. El código se muestra a continuación:

class DIRECTIVAS_MODEL : MonoBehaviour

```
{
    // Parámetros del modelo dinámico del Quadrotor
    public struct PARAM
    {
        public const float ro = 1.2f;    // (Kg/m^3) Densidad
        public const float drag = 9.5e-6f;     // (N*s^2) Coeficiente
de empuje
        public const float cdrag = 1.7e-7f;  // (N*m*s^2) Coeficiente
de arrastre
        public const float l = 3.5f; // (m) Distancia entre el CM y
los rotores
        public const float Ixy = 0.0363f; // (Kg*m^2) Momento de
inercia en x e y
        public const float Iz = 0.0615f; // (Kg*m^2) Momento de
inercia en x e y

        public const float masa = 2.6851f; // (Kg) Masa del dron
    }
    public struct CONS
    {
        public const float wmax = 1047f; // (rad/s) Velocidad angular
máxima
        public const float wmin = 10.5f; // (rad/s) Velocidad angular
mínima
        public const float weq = 762.58624f; // (rad/s) Velocidad
angular que contrarresta la gravedad
        public const float Teq = 22.0579f; // Inversión con el modelo
de weq
        public const float A = 0.0902f;
        public const float B = 0.06461f;
        public const float C = 1.213f;
        public const float D = 0.3679f;
    }

    // Parámetro de entrada del Modelo
    public struct PackModel
    {
        public Rigidbody Quadrotor;
        public Estaticos Static;
    }

    // Variables estáticas del Modelo
```

```
    public struct Estaticos
    {
        public double[] wr1;
        public double[] wr2;
        public double[] wref1;
        public double[] wref2;
    }
}
```

3. DIRECTIVAS_CONTROL: esta clase contiene todo lo necesario para los controladores. Contiene la matriz necesaria para el control de inversión (transformación pares virtuales a velocidades de referencia), las saturaciones para todas las variables a controlar, y el pack del control: este pack utilizado para la comunicación contiene cuatro componentes:
    - Instancia del sólido rígido que se está controlando.
    - Variables estáticas de los controladores.
    - Constantes para los PDs del control de estabilidad.
    - Constantes para los PDs del control de posición.

    Por último nos encontramos con varias estructuras que definen los componentes que se intancian en el pack, como las contantes que contiene un PD (Kp y Kd).

class DIRECTIVAS_CONTROL : MonoBehaviour

```
{
    // Modelo de generación invertido
    public static double[,] FF = {
        {0.25,                                                    0,  -
1/(2*DIRECTIVAS_MODEL.PARAM.l),      DIRECTIVAS_MODEL.PARAM.drag/(4   *
DIRECTIVAS_MODEL.PARAM.cdrag)},
        {0.25,                          1/(2*DIRECTIVAS_MODEL.PARAM.l),
0, -DIRECTIVAS_MODEL.PARAM.drag/(4 * DIRECTIVAS_MODEL.PARAM.cdrag)},
        {0.25,                                                     0,
1/(2*DIRECTIVAS_MODEL.PARAM.l),      DIRECTIVAS_MODEL.PARAM.drag/(4   *
DIRECTIVAS_MODEL.PARAM.cdrag)},
        {0.25,                        -1/(2*DIRECTIVAS_MODEL.PARAM.l),
0, -DIRECTIVAS_MODEL.PARAM.drag/(4 * DIRECTIVAS_MODEL.PARAM.cdrag)}};

    // Saturaciones
    public struct Saturaciones
    {
        public static double rollmax = 10 * CONVERT.G2R; // (rad)
        public static double rollmin = -10 * CONVERT.G2R; // (rad)
        public static double pitchmax = rollmax; // (rad)
        public static double pitchmin = rollmin; // (rad)
    }

    // Variables estáticas del Control
    public struct Estaticos
    {
        public double Tref1;
        public double ex1;
        public double ey1;
        public double ez1;
        public double eroll1;
        public double epitch1;
        public double eyaw1;
```

```
    }

    // Parámetro de entrada del Control
    public struct PackControl
    {
        public Rigidbody Quadrotor;
        public Estaticos Static;
        public STControl P_STControl;
        public POSControl P_PControl;
    }

    // Constantes de un PD
    public struct PD
    {
        public double Kp;
        public double Kd;
    }

    // Constantes de los controladores del STControl
    public struct STControl
    {
        public PD Z;
        public PD Alabeo;
        public PD Cabeceo;
        public PD Guinada;
    }

    // Constantes de los controladores del STControl
    public struct POSControl
    {
        public PD X;
        public PD Y;
    }
}
```

4. DIRECTIVAS_PLAN: esta clase contiene lo necesario para el planificador. En orden, encontramos primero las saturaciones de las velocidades angulares y lineal en z; y la estructura del pack del planificador. Está compuesto por:

    - La instancia al sólido rígido.
    - Una estructura para el planificador de medio nivel. Ésta contiene un vector con los estados de la máquina de estados que gobierna el movimiento (aterrizar, despegar, vigilar…) y una matriz con todas las trayectorias.
    - Una estructura para el planificador de bajo nivel: contiene el tiempo inicial para el movimiento, el punto inicial, y una variable booleana que indica si la trayectoria se está realizando o va a comenzar.

    Por último nos encontramos con varias estructuras que definen los componentes que se intancian en el pack del planificador.

class DIRECTIVAS_PLAN : MonoBehaviour

```
{
    // Saturaciones velocidades
    public struct Saturaciones
    {
        public static double wangmax = 10 * CONVERT.G2R; // (rad/s)
```

```
            public static double wangmin = -10 * CONVERT.G2R; // (rad/s)
            public static double vzmax = 0.5; // (m/s)
            public static double vzmin = -0.5; // (m/s)
        }

        // Parámetro de entrada del Planificador
        public struct PackPlan
        {
            public Rigidbody Quadrotor;
            public MediumLevel MediumLevel;
            public LowLevel LowLevel;
        }

        // Estructura de parámetros necesarios del planificador de medio
nivel
        public struct MediumLevel
        {
            public string Estado;
            public Trayectoria Trayectorias;
        }

        // Estructura de parámetros necesarios del planificador de bajo
nivel
        public struct LowLevel
        {
            public double Tini;
            public Vector4 Pini;
            public bool Start;
        }

        // Estructura que define a una trayectoria
        public struct Trayectoria
        {
            public IdTrayectoria id;
            public Vector4[,] Tray;
        }

        // Estructura que identifica a una trayectoria
        public struct IdTrayectoria
        {
            public string[] Code;
            public int[] NPuntos;
            public int[] Sit;
        }
    }
```

4.2.2 Programa MacroQuad

Este programa es el único que se instancia en cada quadrotor. Esto implica que es el único programa que tiene memoria, dado que tiene que guardar todas las variables que necesitan todos los controladores, el modelo del quadrotor y el planificador, como mínimo.

El programa MacroQuad se compone de los siguientes componentes:

1. Declaración de cuatro variables globales: tres paquetes de comunicación y un vector tipo objeto especial. Los paquetes para el modelo, el control y el planificador; y el objeto especial es un transform. Este tipo de objetos guardan todos las características cinemáticas de un objeto. En concreto, este guardará los transforms de las cuatro hélices, de ahí que sea un vector.

2. Función awake: esta función se iniciará nada más empezar la simulación. Dentro de esta función se hacen llamadas a las siguientes funciones:
    a. IniPacks: esta función da valor a todas las variables de los tres packs de comunicación. En concreto se sigue este orden:
        i. Capta el sólido rígido al que se le ha añadido este código. Este paso es el más importante, dado que las demás funciones actuarán sobre este sólido rígido.
        ii. Definimos un estado para el planificador. Y se le da el valor de *true* al flag de la trayectoria para señalizar que se empezará.
        iii. Se inicializan todas las variables estáticas de los controladores a valor nulo, salvo el par virtual que afecta a la altura que se inicializa al de equilibrio.
        iv. Se le da valor a los parámetros de los PDs de cada control. Se ha decidido ponerlo aquí para que se puedan hacer varios quadrotors cambiando los valores de los controladores que manejan, o cambiar la arquitectura de control y que se pueda inicializar por separado en cada quadrotor.
        v. Por último se inicializan a valor nulo las variables estáticas del modelo.
    b. DefineTrayectorias: esta función define las trayectorias a seguir en cada estado.
    c. IniHelices: esta función capta los transform de cada hélice para luego poder moverlas en la función MueveHelices.
3. Función FixedUpdate: esta función se ejecutará cada fixedTime. Este tiempo es el tiempo cada cual Unity3d realiza todos los cálculos. Este tiempo es distinto al tiempo entre fotogramas, debido a que dependiendo de muchos factores (como el hardware), este fluctuará (bajarán los fps); mientras que el fixedTime siempre será constante. De ahí que es el utilizado para el motor físico. Este tiempo por defecto está definido a 0.02s. Esto aclara lo dicho anteriormente sobre la problemática del tiempo de muestreo entre Matlab y Unity3d.

    Dentro de esta función se implementan cuatro pasos:
    a. Llamada al planificador (QuadPlan.Planificador).
    b. Llamada al control en cascada (QUAD_CONTROL.Control).
    c. Llamada al modelo (QUAD_MODEL.Model).
    d. Llamada a la función MueveHelices: esta función se encarga de mover las hélices modificando su transform directamente, esto implica que no se mueven añadiendo una fuerza, ni un par, dado que toda la dinámica se incluye en el modelo.

A continuación se adjunta el programa MacroQuad:

PROGRAMA MACROQUAD.cs

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

class MacroQuad : MonoBehaviour
{
    // DECLARACIÓN DE VARIABLES GLOBALES
    DIRECTIVAS_MODEL.PackModel PackModel;
    DIRECTIVAS_CONTROL.PackControl PackControl;
    DIRECTIVAS_PLAN.PackPlan PackPlanificador;
    Transform[] Helice = new Transform[4];


    // FUNCIÓN QUE INICIALIZA TODOS LOS COMPONENTES DEL QUADROTOR
    void Awake()
```

```
    {
        IniPacks();
        DefineTrayectorias();
        IniHelices();
    }


    // FUNCIÓN QUE SE EJECUTA CADA FRAME FÍSICO (QUE NO CADA
FOTOGRAMA)
    void FixedUpdate()
    {
        double tiempo = Time.time;
        Vector4 Punto = QuadPlan.Planificador(ref PackPlanificador);
        double[] wref = QUAD_CONTROL.Control(ref PackControl, Punto);
        double[] wr = QUAD_MODEL.Model(ref PackModel, wref);
        MueveHelices(ref Helice, wr);
    }


    // INICIALIZA LOS PACKS DE ENTRADA A LAS FUNCIONES QUE CONTROLAN
CADA DRON
    void IniPacks()
    {
        // Obtenemos el RigidBody que instancia este código
        Rigidbody Quadrotor;
        Quadrotor = GetComponent<Rigidbody>();
        Quadrotor.mass = DIRECTIVAS_MODEL.PARAM.masa;
        Quadrotor.drag = DIRECTIVAS_MODEL.PARAM.drag;
        Quadrotor.angularDrag = DIRECTIVAS_MODEL.PARAM.cdrag;
        Quadrotor.centerOfMass = new Vector3 (0, 0, 0);
        Quadrotor.inertiaTensor = new Vector3
(DIRECTIVAS_MODEL.PARAM.Ixy, DIRECTIVAS_MODEL.PARAM.Iz,
DIRECTIVAS_MODEL.PARAM.Ixy);
        Quadrotor.inertiaTensorRotation = Quaternion.identity;

        // Inicializamos el QuadRotor actual
        PackModel.Quadrotor = Quadrotor;
        PackControl.Quadrotor = Quadrotor;
        PackPlanificador.Quadrotor = Quadrotor;

        // Inicializamos las variables del Planificador
        PackPlanificador.MediumLevel.Estado = "Supervisar";
        PackPlanificador.LowLevel.Start = true;

        // Inicializamos todas las variables estáticas de los
controladores
        PackControl.Static.ex1 = 0;
        PackControl.Static.ey1 = 0;
        PackControl.Static.ez1 = 0;
        PackControl.Static.eroll1 = 0;
        PackControl.Static.epitch1 = 0;
        PackControl.Static.eyaw1 = 0;
        PackControl.Static.Tref1 = DIRECTIVAS_MODEL.CONS.Teq;

        // Inicializamos las constantes de los PDs
        PackControl.P_STControl.Z.Kp = 25.4109;
        PackControl.P_STControl.Z.Kd = 1.3587 *
PackControl.P_STControl.Z.Kp;
```

```
        PackControl.P_STControl.Alabeo.Kp = 0.6815 / 4;
        PackControl.P_STControl.Alabeo.Kd    =    0.3766    *    2    *
PackControl.P_STControl.Alabeo.Kp;
        PackControl.P_STControl.Cabeceo.Kp                              =
PackControl.P_STControl.Alabeo.Kp;                  // Misma función de
transferencia
        PackControl.P_STControl.Cabeceo.Kd                              =
PackControl.P_STControl.Alabeo.Kd;                  // Misma función de
transferencia
        PackControl.P_STControl.Guinada.Kp = 1.2383;
        PackControl.P_STControl.Guinada.Kd      =      0.3022      *
PackControl.P_STControl.Guinada.Kp;
        PackControl.P_PControl.X.Kp = 1.7105;
        PackControl.P_PControl.X.Kd    =    2.2568    *    0.5       *
PackControl.P_PControl.X.Kp;
        PackControl.P_PControl.Y.Kp   =   PackControl.P_PControl.X.Kp;
// Misma función de transferencia
        PackControl.P_PControl.Y.Kd   =   PackControl.P_PControl.X.Kd;
// Misma función de transferencia

        // Inicializamos todas las variables estáticas del modelo
        PackModel.Static.wr1 = new double[4];
        PackModel.Static.wr2 = new double[4];
        PackModel.Static.wref1 = new double[4];
        PackModel.Static.wref2 = new double[4];
        for (int i = 0; i<4; i++)
        {
            PackModel.Static.wr1[i] = 0;
            PackModel.Static.wr2[i] = 0;
            PackModel.Static.wref1[i] = 0;
            PackModel.Static.wref2[i] = 0;
        }
    }


    // FUNCIÓN QUE DEFINE LAS TRAYECTORIAS QUE SE VAN A REALIZAR EN
CADA ESTADO
    void DefineTrayectorias()
    {
        // Acotamos el tamaño de todos los vectores
        PackPlanificador.MediumLevel.Trayectorias.id.Code    =    new
string[3];
        PackPlanificador.MediumLevel.Trayectorias.id.NPuntos   =   new
int[3];
        PackPlanificador.MediumLevel.Trayectorias.id.Sit     =     new
int[3];
        PackPlanificador.MediumLevel.Trayectorias.Tray      =      new
Vector4[1,10];

        // Estado: Supervisar
        PackPlanificador.MediumLevel.Trayectorias.id.Code[0]        =
"Supervisar";
        PackPlanificador.MediumLevel.Trayectorias.id.NPuntos[0] = 10;
        PackPlanificador.MediumLevel.Trayectorias.id.Sit[0] = -1;
        PackPlanificador.MediumLevel.Trayectorias.Tray[0,0]   =   new
Vector4(  0,   0, 8f, Convert.ToSingle(CONVERT.G2R * -90));
```

```
        PackPlanificador.MediumLevel.Trayectorias.Tray[0,1]   =   new
Vector4(  0,   0, 8f, Convert.ToSingle(CONVERT.G2R * -90));
        PackPlanificador.MediumLevel.Trayectorias.Tray[0,2]   =   new
Vector4(-25, -25, 40, Convert.ToSingle(CONVERT.G2R * 0));
        PackPlanificador.MediumLevel.Trayectorias.Tray[0,3]   =   new
Vector4(-25,  25, 40, Convert.ToSingle(CONVERT.G2R * -90));
        PackPlanificador.MediumLevel.Trayectorias.Tray[0,4]   =   new
Vector4( 25,  25, 40, Convert.ToSingle(CONVERT.G2R * 180));
        PackPlanificador.MediumLevel.Trayectorias.Tray[0,5]   =   new
Vector4( 25, -25, 40, Convert.ToSingle(CONVERT.G2R * 90));
        PackPlanificador.MediumLevel.Trayectorias.Tray[0,6]   =   new
Vector4(39.16f,  -55.61f,  42.44f,    Convert.ToSingle(CONVERT.G2R  *  -
90));
        PackPlanificador.MediumLevel.Trayectorias.Tray[0,7]   =   new
Vector4(39.16f, -55.61f, 42.44f,  Convert.ToSingle(CONVERT.G2R * 90));
        PackPlanificador.MediumLevel.Trayectorias.Tray[0,8]   =   new
Vector4(39.16f,  -55.61f,  42.44f,    Convert.ToSingle(CONVERT.G2R  *  -
90));
        PackPlanificador.MediumLevel.Trayectorias.Tray[0,9]   =   new
Vector4(39.16f,  -55.61f,  42.44f,    Convert.ToSingle(CONVERT.G2R  *  -
90));
    }


    // FUNCIÓN QUE BUSCA LAS HÉLICES Y CAPTURA SU TRANSFORM PARA
PODER MOVERLAS
    void IniHelices()
    {
        GameObject QUAD = gameObject;
        Helice[0] = QUAD.transform.Find("DRON").Find("H_1");
        Helice[1] = QUAD.transform.Find("DRON").Find("H_2");
        Helice[2] = QUAD.transform.Find("DRON").Find("H_3");
        Helice[3] = QUAD.transform.Find("DRON").Find("H_4");
    }


    // FUNCIÓN QUE MUEVE LAS HÉLICES
    void MueveHelices(ref Transform[] Helice, double[] wr)
    {
        double t = Time.fixedDeltaTime;
        for (int i = 0; i < 4; i++)
        {
            Helice[i].Rotate(Vector3.up,  Convert.ToSingle(Math.Pow(-
1, i + 1) * CONVERT.R2G * wr[i] * t));
        }
    }
}
```

4.3 Diseño de un modelo

El modelo presentado en la primera parte de este documento, diseñado en el entorno Matlab, no se puede utilizar en Unity3d, dado que es un archivo precompilado.

Para diseñar el modelo se partió de la caja negra que aparece en la figura 2.7, en la cual teníamos un modelo que recibe como entrada las cuatro velocidades de referencia y su salida son todas las variables cinemáticas del quadrotor. Las salidas son fáciles de solucionar, dado que pasando por referencia el sólido rígido que se está modelando, ahí se guardan todos los valores. El modelo interno sí que había que diseñarlo.

En primer lugar se ha diseñado un modelo de segundo orden para la velocidad, como se ha explicado en el capítulo 2.7. Para recordar se muestra a continuación la función de transferencia escogida:

$$G(s) = \frac{w(s)}{wref(s)} = \frac{1}{(0.04s + 1) * (0.04s + 1)}$$

Este modelo de segundo orden se ha discretizado con un tiempo de muestreo de 0.02s en el entorno de Matlab. Como se explicó anteriormente, es el tiempo por defecto en el que se realizan todos los cálculos físicos en Unity3d. Como ya se ha nombrado, los controladores se han verificado para este tiempo de muestreo. El resultado ha sido:

$$wr = A * wref1 + B * wref2 + C * wr1 - D * wr2$$

Siendo A = 0.0902; B = 0.06461; C = 1.213; D = 0.3679. Esto modela un pequeño retraso, que es lo emulado en Matlab como el control de bajo nivel.

Una vez obtenido este modelo discretizado, se programó el código QuadModel.cs, que sigue los siguientes pasos:

1. Una función, *Model,* que es la única que es accesible desde afuera que llama a su vez al modelo. Esta a su vez llama a la función *ModeloDinamico.* Esto se ha realizado de esta manera para que se más fácil cambiar el modelo sin tocar ninguna clase más.
2. La función *ModeloDinamico* implementa el modelo de segundo orden diseñado anteriormente, calcula el vector normal a las hélices, calcula las fuerzas y los pares con estas velocidades siguiendo las siguientes expresiones:

$$fr = drag * ro * wr^2 * n$$

$$taur = cdrag * ro * wr^2 * n$$

   Por último, llama a la función *Transmision* y guarda las variables estáticas en el pack.
3. La función *Transmision* calcula el punto de aplicación correspondiente al centro de cada hélice y aplica las fuerzas y pares calculados anteriormente. Se han implementado dos comprobaciones de seguridad: la primera es para evitar que el valor de la fuerza sea un número imaginario, dado que viene de una raíz cuadrada. Además, el simulador no permite añadir fuerzas nulas. Esto también se comprueba tanto para las fuerzas como para los pares.

A continuación se adjunta el código:

class QUAD_MODEL : MonoBehaviour

```
  {
      public static double[] Model(ref DIRECTIVAS_MODEL.PackModel Pack,
double[] wref)
      {
          // Funciones que modelan al QuadRotor
          double[] wr = ModeloDinamico(ref Pack, wref);
          return wr;
      }

      // FUNCIÓN QUE CONTIENE EL MODELO DINÁMICO PARA LA VELOCIDAD
ANGULAR DE LAS HÉLICES (wref -> wr)
      // Y LAS ECUACIONES DE TRANSFORMACIÓN (Velocidad angular -> Pares
y fuerzas)
      static  double[]  ModeloDinamico(ref  DIRECTIVAS_MODEL.PackModel
Pack, double[] wref)
      {
          // Declaración de variables locales
          double[] wr = new double[4];
          double[,] fr = new double[3,4], taur = new double[3,4];
```

```
        // Modelo dinámico de orden 2 muestreado con un Tm = 0.02s
        for (int i = 0; i < 4; i++)
        {
            wr[i] = DIRECTIVAS_MODEL.CONS.A * Pack.Static.wref1[i]
                  + DIRECTIVAS_MODEL.CONS.B * Pack.Static.wref2[i]
                  + DIRECTIVAS_MODEL.CONS.C * Pack.Static.wr1[i]
                  - DIRECTIVAS_MODEL.CONS.D * Pack.Static.wr2[i];
        }

        // Vector normal a las hélices
        Vector3 n = new Vector3(Pack.Quadrotor.transform.up[0],
Pack.Quadrotor.transform.up[2], Pack.Quadrotor.transform.up[1]);

        // Fuerza producida por las hélices
        for (int j = 0; j < 4; j++)
        {
            for (int i = 0; i < 3; i++)
            {
                fr[i,j] = DIRECTIVAS_MODEL.PARAM.drag *
DIRECTIVAS_MODEL.PARAM.ro * Math.Pow(wr[j],2) * n[i];
            }
        }

        // Par producido por las hélices
        for (int j = 0; j < 4; j++)
        {
            for (int i = 0; i < 3; i++)
            {
                taur[i,j] = Math.Pow(-1,j) *
DIRECTIVAS_MODEL.PARAM.cdrag * DIRECTIVAS_MODEL.PARAM.ro *
Math.Pow(wr[j],2) * n[i];
            }
        }

        // Aplicamos la fuerza y el par
        Transmision(ref Pack, fr, taur);

        // Actualización de las variables
        Pack.Static.wref2 = Pack.Static.wref1;
        Pack.Static.wref1 = wref;
        Pack.Static.wr2 = Pack.Static.wr1;
        Pack.Static.wr1 = wr;

        // Devolvemos la velocidad angular de las hélices
        return wr;
    }

    // FUNCIÓN QUE APLICA LA FUERZA Y EL PAR EN EL QUADROTOR
    static void Transmision(ref DIRECTIVAS_MODEL.PackModel Pack,
double[,] fr, double[,] taur)
    {
        // Declaración de variables
        double[,] Point = new double[3,4];
        Vector4 Centro;
        Vector3 TAU = new Vector3 (0, 0, 0);
        int j, aux;

        // Centro de masas
```

```
        Centro       =       CONVERT.V62V4(CONVERT.TF(Pack.Quadrotor,
CONVERT.NullVector6()));

        // Cálculo del punto de aplicación de las fuerzas
        j = 0;
        aux = 1;
        while (j < 3)
        {
            Point[0,j] = Centro.x + aux    * DIRECTIVAS_MODEL.PARAM.l
* Pack.Quadrotor.transform.right[0];
            Point[1,j] = Centro.y + aux    * DIRECTIVAS_MODEL.PARAM.l
* Pack.Quadrotor.transform.right[2];
            Point[2,j] = Centro.z + aux    * DIRECTIVAS_MODEL.PARAM.l
* Pack.Quadrotor.transform.right[1];
            Point[0,j+1] = Centro.x + aux * DIRECTIVAS_MODEL.PARAM.l
* Pack.Quadrotor.transform.forward[0];
            Point[1,j+1] = Centro.y + aux * DIRECTIVAS_MODEL.PARAM.l
* Pack.Quadrotor.transform.forward[2];
            Point[2,j+1] = Centro.z + aux * DIRECTIVAS_MODEL.PARAM.l
* Pack.Quadrotor.transform.forward[1];
            j += 2;
            aux -= j;
        }

        // Cálculo del par resultante
        for (int i = 0; i < 3; i++)
        {
            for (int k = 0; k < 4; k++)
            {
                TAU[i] += Convert.ToSingle(taur[i,k]);
            }
        }

        // Transmisión final
        bool Flag;
        for (int i = 0; i < 4; i++)
        {
            Flag = true;
            for (int k = 0; k < 3; k++)
            {
                if (Math.Abs(fr[k,i]) > 0 && Flag)
                {
                    Pack.Quadrotor.AddForceAtPosition(new     Vector3
(Convert.ToSingle(fr[0,i]),                 Convert.ToSingle(fr[2,i]),
Convert.ToSingle(fr[1,i])), new Vector3 (Convert.ToSingle(Point[0,i]),
Convert.ToSingle(Point[2,i]), Convert.ToSingle(Point[1,i])));
                    Flag = false;
                }
            }
        }

        Flag = true;
        for (int i = 0; i < 3; i++)
        {
            if (Math.Abs(-TAU[i]) > 0 && Flag)
            {
```

```
                    Pack.Quadrotor.AddRelativeTorque(new    Vector3    (-
TAU[0],-TAU[2],-TAU[1]));
                    Flag = false;
                }
            }

        }
    }
```

4.4 Adaptación del Código

El entorno de Unity3d se programa utilizando el lenguaje de programación C# [8]. Este lenguaje no permite el uso vectorial y matricial utilizado en Matlab. Además hay cambios significativos, como la declaración de variables, los cambios de formatos, etc que también fue necesario adaptar.

Una vez adaptados los códigos diseñados y probados en Matlab de los controladores, se implementó el planificador, utilizando el generador de Splits y creando varios planificadores para distintos niveles de abstracción. Además, el generador de Splits se cambió por el Smart, explicado anteriormente.

Todos los controladores se han implementado en el programa *QuadControl.cs.* Éste, al igual que el modelo, contiene una función accesible desde afuera, *Control*, que llama a todos los controladores en orden.

4.4.1 Adaptación del control de inversión

Una vez cambiadas las diferencias de lenguaje, el resultado final fue el siguiente:

INVERSIÓN DEL MODELO DINÁMICO DEL QUADROTOR

```
    static double[] Inversion(double[] Tauref)
    {
        // Declaración de variables
        double[] Fref = {0, 0, 0, 0};
        double[] wref = {0, 0, 0, 0};

        // Utilizamos el modelo de generación invertido y saturamos
la velocidad
        for (int i = 0; i<4; i++)
        {
            for (int j = 0; j<4; j++)
            {
                Fref[i] += DIRECTIVAS_CONTROL.FF[i,j] * Tauref[j];
            }

            double aux = Fref[i] / DIRECTIVAS_MODEL.PARAM.drag;
            if (aux >= 0)
            {
                wref[i] = Math.Sqrt(aux);
                if (wref[i] > DIRECTIVAS_MODEL.CONS.wmax) wref[i] =
DIRECTIVAS_MODEL.CONS.wmax;
                else   if   (wref[i]   <   DIRECTIVAS_MODEL.CONS.wmin)
wref[i] = DIRECTIVAS_MODEL.CONS.wmin;
            }
            else
            {
                wref[i] = DIRECTIVAS_MODEL.CONS.wmin;;
            }
        }
        return wref;
```

```
    }
```

Como se puede observer, es similar, salvo que la matriz de transformación *FF* se encuentra en otro programa distinto, para ser utilizado más cómodamente. Además, se ha incluido una comprobación para la raíz cuadrada, para evitar valores negativos dentro. De esta comprobación se encarga la variable *aux*.

4.4.2 Adaptación del control de estabilidad

Los controles implementados en el control de estabilidad se modificaron. Anteriormente, la componente derivative de los controladores venían de una transformada de las velocidades y restándolas con las velocidades de referencia, calculadas con una derivada numérica (cfr. capítulo 2.4). Dado que los giros de los ángulos son distintos en Unity3d que en Matlab. Esto hacía que la trnasformación que se aplicaba para saber las velocidades angulares no funcionase correctamente. Para solucionarlo se ha obtenido el error en velocidad como la derivada numérica del error en posición.

Al terminar los cambios dichos anteriormente más las diferencias de código, el resultado fue el siguiente:

CONTROL DE ESTABILIDAD

```
    static double[] ControlEstabilidad(double Tm, ref
DIRECTIVAS_CONTROL.PackControl Pack, CONVERT.Vector6 Ref)
    {
        // Definición de variables
        CONVERT.Vector6 Pos;


////////////////////////////////////////////////////////////
   //////////////   ENTRADAS DEL CONTROLADOR    ////////////////
 ////////////////////////////////////////////////////////////
        // Medidas del Quadrotor:
        Pos = CONVERT.TF(Pack.Quadrotor, Ref);

////////////////////////////////////////////////////////////

        // Implementación del PD de Altura
        double ez = Ref.Z - Pos.Z;
// Error en posicion
        double evz = (ez - Pack.Static.ez1) / Tm;                  //
Derivada numérica del error
        double Ktf = 1 / (Math.Cos(Pos.Roll) * Math.Cos(Pos.Pitch));
        double Tref = DIRECTIVAS_MODEL.CONS.Teq + Ktf *
(Pack.P_STControl.Z.Kp * ez + Pack.P_STControl.Z.Kd * evz);    // weq
se suma a los cuatro motores

        // Implementación del PD del Alabeo (ROLL)
        double eroll = Ref.Roll - Pos.Roll;
// Error en angulo
        double erolld = (eroll - Pack.Static.eroll1) / Tm;         //
Derivada numérica del error
        double Taurollref = Pack.P_STControl.Alabeo.Kp * eroll +
Pack.P_STControl.Alabeo.Kd * erolld;              // wroll positiva
motores +4 2-

        // Implementación del PD del Cabeceo (PITCH)
        double epitch = Ref.Pitch - Pos.Pitch;
// Error en angulo
```

```
        double epitchd = (epitch - Pack.Static.epitch1) / Tm;      //
Derivada numérica del error
        double Taupitchref = Pack.P_STControl.Cabeceo.Kp * epitch +
Pack.P_STControl.Cabeceo.Kd * epitchd;      // wpitch positiva motores
+1 3-

        // Implementación del PD de la Guiñada (YAW)
        double      eyaw      =      Ref.Yaw      -      Pos.Yaw;
// Error en angulo
        double eyawd = (eyaw - Pack.Static.eyaw1) / Tm;            //
Derivada numérica del error
        double  Tauyawref  =  Pack.P_STControl.Guinada.Kp  *  eyaw  +
Pack.P_STControl.Guinada.Kd * eyawd;   // wyaw positiva motores 1 3

        // Actualizamos el valor de las variables estáticas
        Pack.Static.Tref1     = Tref;
        Pack.Static.ez1       = ez;
        Pack.Static.eroll1    = eroll;
        Pack.Static.epitch1   = epitch;
        Pack.Static.eyaw1     = eyaw;

        // Devolvemos los valores de los pares de referencia
        double[] Tauref = {Tref, Taurollref, Taupitchref, Tauyawref};
        return Tauref;
    }
```

4.4.3 Adaptación del control de posición

Para este controlador se han realizado los mismos cambios en los términos derivativos y en el lenguaje de programación:

CONTROL DE POSICIÓN

```
    static    CONVERT.Vector6    ControlPosicion(double    Tm,    ref
DIRECTIVAS_CONTROL.PackControl Pack, CONVERT.Vector6 Ref)
    {
        // Definición de variables
        CONVERT.Vector6 Pos;


////////////////////////////////////////////////////////////
        ////////////////              ENTRADAS    DEL    CONTROLADOR
////////////////

////////////////////////////////////////////////////////////
        // Medidas:
        Pos = CONVERT.TF(Pack.Quadrotor, Ref);

        // Parametros anteriores
        double Tref    = Pack.Static.Tref1;

////////////////////////////////////////////////////////////

        // Implementación del PD en X
        double ex = Ref.X - Pos.X;                          // Error en
posicion
        double  evx  =  (ex  -  Pack.Static.ex1)  /  Tm;    //  Derivada
numérica del error
```

```
        double    Uxref    =    (Pack.P_PControl.X.Kp    *    ex    +
Pack.P_PControl.X.Kd * evx) / Tref;

        // Implementación del PD en Y
        double ey = Ref.Y - Pos.Y;                          // Error en
posicion
        double  evy  =  (ey  -  Pack.Static.ey1)  /  Tm;     //  Derivada
numérica del error
        double    Uyref    =    (Pack.P_PControl.Y.Kp    *    ey    +
Pack.P_PControl.Y.Kd * evy) / Tref;

        // Resolvemos el sistema de ecuaciones
        double Rollref = 0;
        double    Rollaux    =    Math.Sin(Pos.Yaw)    *    Uxref    -
Math.Cos(Pos.Yaw) * Uyref;
        if (Math.Abs(Rollaux) < 1)
        {
            Rollref = Math.Asin(Rollaux);

            // Saturaciones
            if  (Rollref  >  DIRECTIVAS_CONTROL.Saturaciones.rollmax)
Rollref = DIRECTIVAS_CONTROL.Saturaciones.rollmax;
            else              if              (Rollref              <
DIRECTIVAS_CONTROL.Saturaciones.rollmin)           Rollref           =
DIRECTIVAS_CONTROL.Saturaciones.rollmin;
        }
        else
        {
            if (Rollaux < 0)
            {
                Rollref = DIRECTIVAS_CONTROL.Saturaciones.rollmin;
            }
            else
            {
                Rollref = DIRECTIVAS_CONTROL.Saturaciones.rollmax;
            }
        }
        double Pitchref = 0;
        double    Pitchaux    =    (Math.Cos(Pos.Yaw)    *    Uxref    +
Math.Sin(Pos.Yaw) * Uyref) / (Math.Cos(Rollref));
        if (Math.Abs(Pitchaux) < 1)
        {
            Pitchref = Math.Asin(Pitchaux);

            // Saturaciones
            if  (Pitchref  >  DIRECTIVAS_CONTROL.Saturaciones.pitchmax)
Pitchref = DIRECTIVAS_CONTROL.Saturaciones.pitchmax;
            else              if              (Pitchref              <
DIRECTIVAS_CONTROL.Saturaciones.pitchmin)           Pitchref           =
DIRECTIVAS_CONTROL.Saturaciones.pitchmin;
        }
        else
        {
            if (Pitchaux < 0)
            {
                Pitchref = DIRECTIVAS_CONTROL.Saturaciones.pitchmin;
            }
```

```
            else
            {
                Pitchref = DIRECTIVAS_CONTROL.Saturaciones.pitchmax;
            }
        }

        // Actualizamos las variables estáticas
        Pack.Static.ex1 = ex;
        Pack.Static.ey1 = ey;

        // Devolvemos los valores de referencia
        Ref.Roll = Convert.ToSingle(Rollref);
        Ref.Pitch = Convert.ToSingle(Pitchref);
        return Ref;
    }
```

4.4.4 Adaptación del generador y creación del planificador

Se ha diseñado un planificador para que todo el análisis sensorial lo hiciese éste. En éste se ha incluido el generador de splits Smart. Se ha diseñado por niveles, como se verá a continuación.

El programa que implementa el planificador es el programa *QuadPlan.cs*. EL algoritmo que sigue se compone de los siguientes elementos:

1. Una función externa que llama a los distintos niveles del planificador. En este caso solo se han diseñado dos: el nivel medio y el bajo.
2. Nivel medio: este nivel se encarga de ir leyendo las trayectorias e ir gestionándolas, si empiezan o acaban, etc. Al final devuelve un punto objetivo para el nivel bajo.
3. Nivel bajo: este nivel gestiona el generador de Splits y averigua si se ha alcanzado el punto objetivo o no, para avisar al nivel medio para que le de otro objetivo. Esta comprobación se realiza leyendo el tiempo de movimiento.
4. El generador de Splits es similar al diseñado en Matlab, cambiando los siguientes aspectos:
    a. El lenguaje de programación.
    b. Se ha implementado un algoritmo sencillo para que el tiempo del movimiento sea calculado dentro del generador de Splits (el tiempo de movimiento es proporcional a la distancia).

A continuación se expone el código resultante del planificador:

PLANIFICADOR

```
class QuadPlan : MonoBehaviour
{
    // FUNCIÓN MAESTRA
    public static Vector4 Planificador(ref DIRECTIVAS_PLAN.PackPlan
Pack)
    {
        // Declaración del tiempo de muestreo
        double tiempo = Time.time;

        // Llamadas a los planificadores de más nivel a menos
        Vector4 PuntoObj = MediumLevel(ref Pack);
        return LowLevel(ref Pack, PuntoObj, tiempo);
    }


    // PLANIFICADOR DE MEDIO NIVEL
    static Vector4 MediumLevel(ref DIRECTIVAS_PLAN.PackPlan Pack)
```

```
    {
        // Declaración de variables
        int IdTray = 0;

        // Algoritmo de búsqueda de la trayectoria
        while (Pack.MediumLevel.Estado != Pack.MediumLevel.Trayectorias.id.Code[IdTray])
        {
            IdTray++;
        }

        // Búsqueda de la posición siguiente
        if (Pack.MediumLevel.Trayectorias.id.Sit[IdTray] < Pack.MediumLevel.Trayectorias.id.NPuntos[IdTray] - 1)
        {
            if (Pack.LowLevel.Start == true) Pack.MediumLevel.Trayectorias.id.Sit[IdTray]++;
        }
        else
        {
            Pack.MediumLevel.Trayectorias.id.Sit[IdTray] = 2;
        }
        // Devolvemos el punto objetivo siguiente dentro de esa trayectoria
        return Pack.MediumLevel.Trayectorias.Tray[IdTray, Pack.MediumLevel.Trayectorias.id.Sit[IdTray]];
    }

    // PLANIFICADOR DE BAJO NIVEL
    static Vector4 LowLevel(ref DIRECTIVAS_PLAN.PackPlan Pack, Vector4 PuntoObj, double tiempo)
    {
        // Definición de variables
        double Tmov = 0;
        Vector4 Point;

        // Si es la primera vez, inicializamos la trayectoria puntual
        if (Pack.LowLevel.Start)
        {
            Pack.LowLevel.Start = false;
            Pack.LowLevel.Tini = tiempo + 1;
            Pack.LowLevel.Pini = CONVERT.V62V4(CONVERT.TF(Pack.Quadrotor, CONVERT.NullVector6()));
        }

        // Llamamos al generador de Splits y verificamos si se ha acabdo el movimiento
        Point = GSplits(Pack.LowLevel.Pini, PuntoObj, Pack.LowLevel.Tini, ref Tmov, tiempo, 1);
        if (tiempo > (Tmov + Pack.LowLevel.Tini + 1))
        {
            Pack.LowLevel.Start = true;
        }

        // Devolvemos el punto devuelto por el generador de splits
        return Point;
    }
```

```
      // GENERADOR DE LOS SPLITS ENTRE LOS PUNTOS DEL PLANIFICADOR
      static Vector4 GSplits(Vector4 Pini, Vector4 Pfin, double t_ini,
ref double t_mov, double t, double Kt)
      {
          // Variable locales
          Vector4 Punto = Pini;

          // Calculamos el tiempo del movimiento
          double distmax = 0;
          for (int i = 0; i<4; i++)
          {
              if ((i < 3) && (Math.Abs(Pfin[i] - Pini[i]) > distmax))
distmax = Math.Abs(Pfin[i] - Pini[i]);
              else if ((i == 3) && (Math.Abs(Pfin[i] - Pini[i]) *
CONVERT.R2G / 6 > distmax)) distmax = Math.Abs(Pfin[i] - Pini[i]) *
CONVERT.R2G / 6;
          }
          t_mov = Kt * distmax;
          if (distmax > 10) t_mov /= 2;

          // Obtenemos los puntos intermedios
          if (t > t_mov + t_ini) // Tramo final
          {
              for (int i = 0; i<4; i++) Punto[i] = Pfin[i];
          }
          else if (t > (t_mov/2 + t_ini))
          {
              for (int i = 0; i<4; i++) Punto[i] = Convert.ToSingle((1
- 2*Math.Pow(((t - t_ini - t_mov)/t_mov), 2)) * (Pfin[i] - Pini[i]) +
Pini[i]);
          }
          else if (t > t_ini)
          {
              for    (int    i    =    0;    i<4;    i++)    Punto[i]    =
Convert.ToSingle((2*Math.Pow(((t  -  t_ini)/t_mov),  2))  *  (Pfin[i]  -
Pini[i]) + Pini[i]);
          }

          // Devolvemos el punto destino
          return Punto;
      }
  }
```

5 CONCLUSIÓN

Este proyecto ha consistido en el diseño de una arquitectura de control y sus controladores internos en el entorno de Matlab; posteriormente se ha diseñado un modelo geométrico en 3d en la herramienta Cinema4D; y en último lugar se han juntado todos estos componentes en un simulador virtual con un motor físico que le aporta un carácter de similitud con la realidad.

Este simulador puede usarse para diferentes apliciones de *Gemelos digitales*, como puede ser el control predictivo o las pruebas de quadrotors.

El siguiente paso de este proyecto será la mejora de los controladores, la adición de varios niveles más del planificador, la complicación del modelo dinámico diseñado para el quadrotor, la inclusión en el simulador de un entorno para dar aplicación a este quadrotor o más de uno, para posteriormente pasarlo a realidad virtual.

Dada su arquitectura modular y escalable, este paso será bastante sencillo.

BIBLIOGRAFÍA

[1] Mario Haberle, 2016. Vídeo explicativo de dónde se ha obtenido la figura.

URL: https://www.youtube.com/watch?v=3R_V4gqTs_I&list=PLPAgqhxd1Ib1YYqYnZioGyrSUzOwead17&index=1. Fecha de consulta: 4 de Julio de 2020.

[2] Erik Nordeus, 2016. Vídeo explicativo de dónde se ha información sobre un proyecto ya realizado.

URL: https://www.habrador.com/tutorials/pid-controller/3-stabilize-quadcopter/. Fecha de consulta: 4 de Julio de 2020.

[3] Adolfo J. sanchez, 2012. "Simulador para control y automatización utilizando un entorno virtual 3D interactivo y configurable".

[4] The MathWorks. Página oficial del programa Matlab.

URL: https://es.mathworks.com/products/matlab.html. Fecha de consulta: 5 de Julio de 2020.

[5] Luis Llamas, 2016. Web de dónde se ha obtenido la figura.

URL: https://www.luisllamas.es/medir-la-inclinacion-imu-arduino-filtro-complementario/. Fecha de consulta: 5 de julio de 2020.

[6] Josué David Moreno, 2019. ¿Qué es Cinema 4D?

URL: https://www.calamoycran.com/blog/que-es-cinema-4d. Fecha de consulta: 20 de Abril de 2020.

[7] Unity Documentation. Descargando e Instalando Unity, 2016.

URL: https://docs.unity3d.com/es/530/Manual/InstallingUnity.html. Fecha de consulta: 23 de Mayo de 2020.

[8] José Antonio González Seco, 2001. El lenguaje de programación C#.

URL: https://programacion.net/articulo/el_lenguaje_de_programacion_c_167/3. Fecha de consulta: 30 de Mayo de 2020.